

Java学习群

72030155

进群可以免费获取视频教程以及每日免费听老师讲课

Eclipse插件开发 学习笔记

张鹏 姜昊 许力 编著



电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书由浅入深、有重点、有针对性地介绍了Eclipse插件开发技术，全书分为4篇共24章。第一篇介绍Eclipse平台界面开发的基础知识，包括SWT控件的使用、界面布局、事件处理等内容；第二篇是插件开发核心技术，主要介绍插件开发的核心知识要点，包括行为(Action)、视图(ViewPart)、编辑器(Editor)、透视图(Perspective)等10章的内容；第三篇主要讲述插件开发的高级内容，包括开发高级内容、富客户端平台技术(RCP)、Draw2d，以及GEF介绍与实现等4个章节；第四篇则围绕插件开发和GEF应用两个主题，精心设计了两个程序开发实例，使读者能更加全面地理解插件开发相关技术。

本书内容全面，讲解仔细，不仅适合没有Eclipse平台技术基础的相关人士，也适合了解相关技术、具有一定插件开发能力的读者。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目(CIP)数据

Eclipse插件开发学习笔记 / 张鹏, 姜昊, 许力编著. 北京: 电子工业出版社, 2008.7
ISBN 978-7-121-05498-3

I. E… II. ①张…②姜…③许… III. 软件工具—程序设计 IV. TP311.56

中国版本图书馆CIP数据核字(2007)第185044号

责任编辑: 高洪霞

印 刷: 北京天宇星印刷厂

装 订: 三河市皇庄路通装订厂

出版发行: 电子工业出版社

北京市海淀区万寿路173信箱 邮编100036

开 本: 860×1092 1/16 印张: 41.75 字数: 1050千字 彩插: 4

印 次: 2008年7月第1次印刷

印 数: 5000册 定价: 75.00元(赠光盘1张)

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至zlts@phei.com.cn, 盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线: (010) 88258888。



本书独一无二的优势

本书由多位具有丰富一线开发经验的资深程序设计师经过半年多的酝酿、写作、修改而完成。写作手法流畅而不失细腻，直白而不乏生动，既有精要的知识点分析，也有翔实的代码点评，从各个方面将Eclipse插件开发的技术进行了完整直观的展现。

本书的内容跳出了常规介绍Eclipse书籍的套路，围绕插件开发这个中心来展开讲解，同时考虑了读者的基础，有重点地加入了SWT和JFace等方面的内容，使得没有这方面基础的读者也可以迅速掌握插件开发的基础知识。本书的插件开发核心技术和高级进阶部分覆盖了插件开发中的大部分常用技术，其中不乏对相关其他技术的介绍，这样不仅介绍了插件的知识，更将解决问题的思路、方法及过程展现给读者，而这也正是本书要达到的最大目的。

本书的一大特色是图文并茂的叙述方式，通过各种样式丰富的图形，辅以精要的文字注释与说明，直观地进行知识点的分析和讲解，大大提高了读者的阅读兴趣和理解、掌握知识的速度。兴趣是最好的老师，而丰富多样的表现形式则能大大激发读者的学习兴趣，相信本书的努力能获得广大读者的认同。本书和市场上其他类似书籍相比，具有下面与众不同的特色。

形象

本书对书中的源代码采取绘图的讲解方式，使抽象变形象，让高深的编程理论“赤裸裸”地裸露在你的面前，使你轻而易举地领悟编程奥秘。这是本书最大的特色。

趣味

本书用拉链做素材，拉开学习的帷幕，用情趣似的学习笔记形式带你走进轻松的学习课堂，并且在书中让人感觉到烦躁的地方搭配以相关的对应图片，让你享受到学习中的乐趣，探索中的喜悦，胜利后的兴奋。

实用

本书的实用性较强，以经验为后盾，以实践为导向，以实用为目标，深入浅出地讲解Eclipse插件开发中的种种问题。特别是，在讲解时非常注重实践与理论的形象结合，为了使读者通过读懂源代码来掌握Eclipse插件开发技术，在实例举证时还给源代码绘制了大量的流程图。本书的所有源代码都已调试通过，并且放在了本书所附带的光盘中，读者拿来即可使用。

全面

本书内容全面，从基本的语法入手，以恰当的实例为导向，由浅入深地讲解Eclipse插件开发技术的基本理论知识，所讲解的内容几乎囊括了Eclipse插件开发技术的所有知识点。

Eclipse插件开发学习笔记



目录 Content



第一篇 开发基础

第1章 Eclipse平台简介 01

- 1.1 Eclipse集成开发环境(IDE)介绍 02
 - 1.1.1 安装及使用Eclipse IDE 02
 - 1.1.2 为Eclipse IDE安装中文语言包 06
 - 1.1.3 IDE的环境设置 07
 - 1.1.4 使用帮助系统 09
- 1.2 什么是Eclipse 09
- 1.3 SWT/JFace技术 11
- 1.4 插件技术和OSGi 12
- 1.5 RCP技术 15
- 1.6 EMF技术 16
- 1.7 GEF技术 17
- 1.8 本章小结 18

第2章 SWT/JFace概述 19

- 2.1 SWT结构浅析 20
- 2.2 SWT API结构 21
 - 2.2.1 组件类 22
 - 2.2.2 布局类 23
 - 2.2.3 事件类 23
 - 2.2.4 图形类与系统资源管理 25
 - 2.2.5 其他内容 27
- 2.3 JFace 27
 - 2.3.1 查看器 27
 - 2.3.2 资源注册表 28
 - 2.3.3 字段帮助 29
 - 2.3.4 操作和贡献 29
 - 2.3.5 对话框、向导页和偏好设置 30
 - 2.3.6 数据绑定 30
- 2.4 SWT与Swing 31
- 2.5 编写并发布SWT程序 33
 - 2.5.1 第一个SWT程序 33
 - 2.5.2 SWT程序的打包发布 37
- 2.6 本章小结 38

第3章 SWT编程基础 39

- 3.1 Display和Shell 40
 - 3.1.1 Display的创建 40

- 3.1.2 Shell的创建 42
- 3.1.3 Display的事件队列和事件循环 43
- 3.1.4 Display的生命周期 46
- 3.1.5 监视器、边界和客户区域 47
- 3.1.6 SWT程序中的多线程 48

3.2 控件 50

- 3.2.1 控件类的继承结构 50
- 3.2.2 控件的构造和样式 52
- 3.2.3 控件的继承检查 53
- 3.2.4 控件的用户数据 55
- 3.2.5 控件的释放 55

3.3 图形资源 56

- 3.3.1 使用Color 56
- 3.3.2 使用Image 57
- 3.3.3 使用Font 59

3.4 高级内容 60

- 3.4.1 使用系统托盘 60
- 3.4.2 利用Region构造不规则窗口 61
- 3.4.3 在SWT中使用Swing 62

3.5 本章小结 63

第4章 使用基本控件与对话框 64

- 4.1 Button 65
- 4.2 Label 66
- 4.3 Text 68
- 4.4 List 70
- 4.5 Combo 73
- 4.6 ToolBar 和ToolItem 74
- 4.7 Menu和MenuItem 78
- 4.8 CoolBar和CoolItem 81
- 4.9 TabFolder和TabItem 83
- 4.10 对话框 85
 - 4.10.1 消息框 85
 - 4.10.2 文件与目录对话框 87
 - 4.10.3 颜色对话框 89
 - 4.10.4 字体对话框 90
- 4.11 本章小结 91

第5章 容器与布局管理器 92

- 5.1 Composite 93

5.2 Group	96
5.3 Shell	97
5.4 容器上下文菜单设置	100
5.5 容器颜色、背景和鼠标指针设置	102
5.6 布局管理概述	104
5.6.1 AbsoluteLayout布局 (No Layout)	104
5.6.2 FillLayout布局	105
5.6.3 RowLayout布局	106
5.6.4 GridLayout布局	110
5.6.5 FormLayout布局	112
5.6.6 StackLayout布局	114
5.6.7 布局的选择规则	116
5.6.8 自定义布局管理器	116
5.7 本章小结	120
第6章 界面开发工具	121
6.1 安装Visual Editor	122
6.2 使用Visual Editor	123
6.2.1 Visual Editor的基本使用	123
6.2.2 控件布局	128
6.2.3 运行与调试	130
6.3 其他工具介绍	133
6.3.1 SWT Designer	133
6.3.2 Jigloo	133
6.4 本章小结	134
第7章 高级控件使用	135
7.1 列表、表格和树	136
7.1.1 查看器 (Viewer) 框架	136
7.1.2 JFace 列表查看器(List Viewer)	139
7.1.3 Table控件	143
7.1.4 JFace 表格查看器(Table Viewer)	147
7.1.5 Tree控件	148
7.1.6 JFace树查看器 (Tree Viewer)	151
7.2 文本编辑器	155
7.3 滚动条、Scrollable、ScrolledComposite 和滑动条	158
7.4 进度条与进度指示器	160
7.5 浏览器与OLE	162
7.6 本章小结	165
第8章 SWT/Jface的事件处理	166
8.1 SWT的事件处理	167
8.1.1 事件处理机制	167
8.1.2 低级事件类	170
8.1.3 高级事件类	171

8.2 常用事件	172
8.2.1 鼠标事件	172
8.2.2 键盘事件	173
8.2.3 Paint事件	174
8.2.4 应用举例	175
8.2.5 使用SWT模拟键盘/鼠标事件	177
8.3 JFace事件处理	178
8.3.1 操作 (Action) 与贡献(Contribution)	178
8.3.2 创建操作	180
8.3.3 使用贡献	181
8.4 本章小结	182

第二篇 核心技术

第9章 Eclipse插件体系结构	183
9.1 Eclipse体系结构	184
9.1.1 Eclipse平台架构	184
9.1.2 插件工作模式	185
9.1.3 工作台层次结构	186
9.2 插件的加载过程	187
9.2.1 插件的安装	187
9.2.2 插件的发现和启动	190
9.2.3 插件信息的获取	190
9.3 插件的扩展模式	191
9.3.1 扩展和扩展点	191
9.3.2 扩展加载过程	194
9.3.3 常用扩展点	194
9.3.4 小结	195
9.4 本章小结	195
第10章 开发第一个插件项目	196
10.1 创建插件工程	197
10.1.1 新建插件	197
10.1.2 使用向导	198
10.2 “插件开发”透视图	200
10.2.1 PDE视图	201
10.2.2 PDE运行时视图	202
10.2.3 清单编辑器	203
10.3 插件工程结构	203
10.4 插件文件	204
10.4.1 Plugin.xml文件	204
10.4.2 MANIFEST.MF文件	205
10.4.3 Build.properties文件	206
10.5 插件类	207
10.6 运行插件程序	208

10.7 调试插件	210	12.3 创建一个视图	248
10.8 发布插件	211	12.3.1 添加category	248
10.9 本章小结	212	12.3.2 在plugin.xml中声明视图	249
第11章 操作 (Actions)	213	12.4 视图类	250
11.1 Eclipse中的操作概览	214	12.4.1 视图方法	250
11.2 添加工作台窗口操作	214	12.4.2 视图控制	250
11.2.1 使用模板创建扩展	215	12.4.3 视图模型	252
11.2.2 定制操作集	217	12.4.4 视图内容	255
11.2.3 定制工作台菜单	218	12.4.5 视图标签	257
11.2.4 定制操作菜单项	219	12.4.6 视图排序	257
11.2.5 实现操作代理类	221	12.4.7 视图过滤	259
11.3 IAction与IActionDelegate接口	222	12.5 为视图添加操作	260
11.4 对象操作	224	12.5.1 视图选择	260
11.4.1 添加对象操作	225	12.5.2 添加/删除操作	261
11.4.2 操作的可见性	227	12.5.3 过滤操作	263
11.4.3 操作的过滤	228	12.5.4 快捷键支持	265
11.4.4 实现IObjectActionDelegate接口	228	12.6 视图间通信	265
11.5 视图操作	230	12.6.1 对属性视图提供支持	265
11.5.1 添加视图的上下文菜单	231	12.6.2 共享并监听地址本视图查看器	266
11.5.2 添加视图的工具栏	232	12.6.3 提供显示在Properties视图中的内容	266
11.5.3 添加视图的下拉子菜单	232	12.6.4 监听其他Workbench部分被选中的内容	271
11.5.4 实现IViewActionDelegate接口	233	12.7 添加状态栏支持	272
11.6 编辑器操作	234	12.8 视图状态	273
11.6.1 创建编辑器上下文操作	234	12.8.1 存储排序和过滤信息	273
11.6.2 添加顶层工作台菜单	234	12.8.2 存储视图元素信息	277
11.6.3 定义编辑器顶层操作	235	12.9 加载和卸载图标	279
11.6.4 添加编辑器工具栏操作	235	12.10 本章小结	281
11.6.5 实现IEditorActionDelegate接口	236	第13章 编辑器 (Editors)	282
11.7 快捷键映射	237	13.1 Eclipse编辑器体系结构概览	283
11.7.1 键绑定的策略	237	13.2 Eclipse工作环境中的编辑器	284
11.7.2 创建命令	237	13.2.1 AbstractTextEditor类	285
11.7.3 键绑定	238	13.2.2 MultiEditor类	286
11.7.4 命令与操作关联	239	13.2.3 MultiPageEditorPart类	287
11.8 本章小结	240	13.2.4 FormEditor类	287
第12章 视图 (Views)	241	13.3 为例子增加一个编辑器	289
12.1 Eclipse视图体系结构概览	242	13.3.1 声明编辑器	289
12.2 Eclipse工作环境中的视图	243	13.3.2 创建编辑器	290
12.2.1 资源导航	244	13.3.3 创建编辑器输入	291
12.2.2 PageBook视图	245	13.3.4 关联编辑器与编辑器输入	292
12.2.3 大纲视图	245	13.4 编辑器使用的数据模型	294
12.2.4 属性视图	246	13.4.1 基本数据模型	294
12.2.5 任务视图和书签视图	247	13.4.2 数据管理模型	299
12.2.6 项目资源管理器	247	13.4.3 文件映射模型	300
		13.5 编辑器页面	301

13.5.1 “编辑”页	302
13.5.2 “分页预览”页	311
13.5.3 “源代码”页	312
13.6 响应编辑器更改	313
13.6.1 更新模型	313
13.6.2 使“分页预览”页支持更新	317
13.7 保存编辑器模型	318
13.8 编辑器生命周期	322
13.8.1 已修改的编辑器	322
13.8.2 切换页面	324
13.8.3 保存内容	324
13.9 为编辑器添加操作	326
13.9.1 上下文菜单	326
13.9.2 管理编辑器操作栏	329
13.10 本章小结	333
第14章 透视图 (Perspectives)	334
14.1 什么是透视图	335
14.2 创建一个透视图	336
14.2.1 透视图扩展点	336
14.2.2 创建透视图工厂类	338
14.3 IPageLayout	339
14.4 填充透视图	341
14.5 扩展现有透视图	344
14.5.1 添加视图和占位符	345
14.5.2 添加快捷方式	347
14.6 本章小结	348
第15章 对话框和向导 (DialogsWizards)	349
15.1 对话框和向导概述	350
15.2 对话框类别	350
15.2.1 SWT对话框	351
15.2.2 JFace对话框	352
15.2.3 常用JFace对话框	353
15.3 为例子增加SWT对话框	354
15.4 创建JFace对话框	355
15.4.1 使用Dialog类创建JFace对话框	355
15.4.2 为“地址本”视图创建过滤器对话框	356
15.5 向导介绍	362
15.5.1 向导对话框	363
15.5.2 向导	363
15.5.3 向导页面	364
15.6 添加向导	364

15.6.1 定义向导扩展	365
15.6.2 实现向导	366
15.6.3 实现向导页面	368
15.6.4 添加向导处理逻辑	371
15.7 本章小结	378
第16章 首选项 (Preferences)	379
16.1 首选项页面结构	381
16.2 添加首选项页面	382
16.3 示例首选项	383
16.3.1 示例首选项页面	383
16.3.2 字段编辑器	385
16.4 为例子创建首选项页面	387
16.4.1 创建根首选项页面	387
16.4.2 创建“视图”子首选项页面	388
16.4.3 创建“编辑器”子首选项页面	391
16.5 本章小结	395
第17章 帮助内容 (Help Contents)	397
17.1 Eclipse帮助介绍	398
17.2 使用Eclipse帮助	399
17.3 实现集成的帮助文档	400
17.3.1 添加帮助内容扩展	400
17.3.2 添加帮助内容	402
17.4 上下文相关帮助	404
17.4.1 声明帮助的上下文扩展	404
17.4.2 定义弹出信息内容	405
17.4.3 关联弹出信息同UI上下文	406
17.5 加入帮助按钮	408
17.6 本章小结	409
第18章 备忘单 (CheatSheet)	410
18.1 使用Eclipse备忘单	412
18.2 为地址本插件创建备忘单	413
18.3 构建复合备忘单	418
18.4 链接备忘单到帮助	421
18.5 本章小结	424
第三篇 高级进阶	
第19章 插件开发高级内容	426
19.1 自定义扩展点	427
19.1.1 扩展点基础知识	427
19.1.2 使用扩展点	435
19.2 插件的国际化支持	441
19.2.1 国际化方法	441

19.2.2 国际化支持和段 (fragment)	455	22.1.1 GEF架构概述	528
19.2.3 外部化plugin.xml中的字符串	458	22.1.2 GEF工作原理	530
19.3 使用功能部件(feature)	460	22.2 理解GEF-MVC架构	532
19.3.1 功能部件概述	460	22.2.1 GEF中的模型	532
19.3.2 创建功能部件	461	22.2.2 GEF中的视图	533
19.3.3 更新站点与自动更新功能	463	22.2.3 GEF中的控制器 (EditPart)	535
19.3.4 添加产品标签 (branding)	467	22.3 GEF中的基本概念	537
19.4 动态插件	469	22.3.1 Request和Command	537
19.5 本章小结	472	22.3.2 RootEditParts	541
第20章 富客户端平台(RCP)技术	473	22.3.3 EditPolicy和 Role	542
20.1 富客户端技术介绍	474	22.3.4 图形视图 (GraphicalViewer)	546
20.2 RCP平台架构	477	22.3.5 EditDomain和CommandStack	547
20.2.1 RCP结构概述	477	22.3.6 调色板(Palette)和工具(Tool)	550
20.2.2 RCP程序的结构	480	22.3.7 行为(Action)	554
20.3 开发RCP产品	484	22.3.8 坐标系统与层次 (Layer)	554
20.3.1 创建RCP工程	484	22.4 创建GEF应用	557
20.3.2 发布RCP工程	489	22.4.1 添加模型	559
20.4 将插件改造成RCP程序	494	22.4.2 添加视图	560
20.4.1 为RCP添加帮助内容	496	22.4.3 创建控制中心 (EditPart)	561
20.4.2 RCP程序的自动更新	502	22.4.4 创建编辑器 (Editor)	564
20.4.3 为程序添加欢迎页面 (Intro)	504	22.4.5 添加调色板 (palette)	567
20.5 本章小结	508	22.4.6 添加连接线 (Connections)	569
第21章 Draw2d	509	22.4.7 添加属性视图	572
21.1 理解Draw2d	510	22.4.8 添加大纲视图 (Outline)	575
21.1.1 Draw2d系统组成	510	22.4.9 实现拖放功能 (Drag and drop)	578
21.1.2 LightweightSystem简介	511	22.4.10 放大缩小 (Zooming)	579
21.2 Draw2D Figure简介	514	22.4.11 添加直接编辑功能	581
21.3 坐标系统	516	22.4.12 其他相关内容	583
21.4 布局管理	517	22.5 本章小结	585
21.4.1 布局管理器	517		
21.4.2 布局更新	518	第四篇 综合实例	
21.5 连线和路由	520	第23章 插件开发实例	586
21.5.1 连线	520	23.1 需求分析与设计	587
21.5.2 连线路由器 (Connection Router)	520	23.1.1 需求整理	587
21.6 Draw2d中的其他组件	521	23.1.2 数据模型抽象	587
21.6.1 边界	521	23.1.3 体系架构和插件结构	591
21.6.2 层次	522	23.2 插件开发	594
21.6.3 定位器	523	23.2.1 插件rcpdev.thirdparty	594
21.6.4 连接锚点	523	23.2.2 插件rcpdev.common.core和 rcpdev.common.ui	595
21.7 本章小结	525	23.2.3 插件rcpdev.todo.core	602
第22章 GEF介绍与实现	526	23.2.4 插件rcpdev.todo.ui	604
22.1 GEF简介	527	23.2.5 插件rcpdev.todo.persistence	613
		23.3 在RCP程序中重用插件	614

23.3.1 Contact插件介绍	615
23.3.2 查看器的排序	617
23.4 FAQ	622
23.5 本章小结	629
第24章 GEF实例	630
24.1 设计思路	631
24.2 创建项目	631
24.3 创建Editor	635
24.4 构建模型	639
24.5 构建控制器	640
24.6 创建Figure	644
24.7 交互处理	649
24.8 创建调色板	654
24.9 本章小结	656

第1章 Eclipse入门

拉开崭新的学习帷幕

在学习插件开发技术之前，首先要了解Eclipse平台。Eclipse近年来声名鹊起，很大程度上是作为一个Java IDE而广为人知，但这只是它的一个用途而已。除了作为Java开发工具外，使用这个平台还可以完成许多其他工作，了解这些内容，是进一步学习插件开发的基础。通过本章的学习，读者将熟悉Eclipse开发环境的基本使用，并对Eclipse平台的体系架构以及平台上的主要技术有所了解。

本章内容包括：

- ★Eclipse集成开发环境介绍。
- ★什么是Eclipse。
- ★SWT/JFace技术。
- ★插件技术和OSGi。
- ★RCP技术。
- ★EMF技术。
- ★GEF技术。



进入第01章

1.1 Eclipse集成开发环境 (IDE) 介绍

为了方便初次接触Eclipse的读者学习, 本节将对如何下载、安装并使用Eclipse IDE进行简单的介绍, 如果读者已经很熟悉Eclipse的日常使用, 可以跳过这一节。以下的介绍都以Windows XP操作系统平台为例, 并假设读者已经对Java程序开发有一定的了解。

1.1.1 安装及使用Eclipse IDE

运行Eclipse IDE前, 首先需要安装Java运行时环境 (JRE)。最新版本的JRE可以在java.sun.com下载。在Eclipse网站 (<http://www.eclipse.org/downloads/>) 可以下载最新版本的Eclipse IDE, 截至完稿前, Eclipse IDE最新的稳定版本是3.2.2版。

下载完成后, 将得到的压缩包解压到任意目录下, 得到一个名为eclipse的文件夹。运行其中的eclipse.exe启动IDE, 在显示一个欢迎图片后, 如果是第一次运行, Eclipse会要求用户指定一个目录作为工作空间 (workspace), 如图1-1所示。这个目录将存放生成的所有Java项目代码和配置。

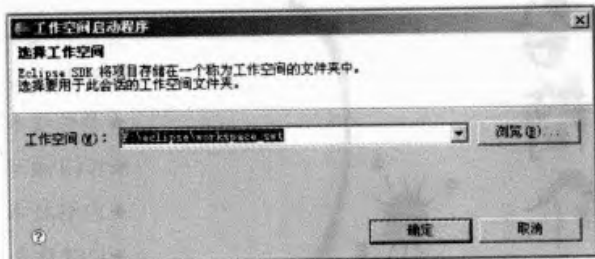


图1-1 选择工作空间

选择工作空间后, 将看到如图1-2所示的欢迎界面, 这个页面为初次接触Eclipse的用户提供了基础知识的介绍。将鼠标指向其中的大图标, 会显示其内容。单击便可进入相应的介绍页面。

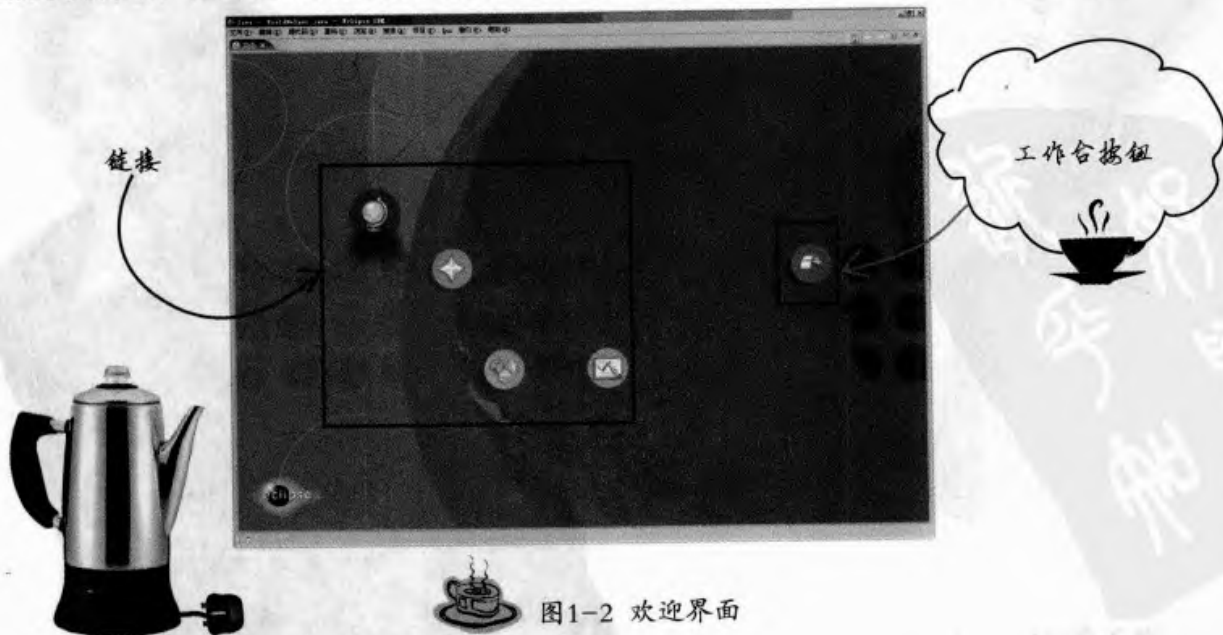


图1-2 欢迎界面

在欢迎界面中，有4个链接，单击它们就可以进入对应的介绍页面。概述包含了面向Eclipse新用户，对整个Eclipse IDE的简单介绍；新增内容对这一个版本的Eclipse IDE新添加的功能进行说明；样本包含了循序渐进的数个例子程序（可能需要连接到Eclipse网站下载）；而教程则将带用户一步一步开始Eclipse的开发之旅。建议读者在开始正式使用Eclipse之前，首先将这些内容通读一遍，对后面的学习将有很大帮助。浏览完介绍内容后，单击最右面的工作台按钮，将看到如图1-3所示的界面。这就是Eclipse的主界面，即被称为工作台（Workbench）的部分，日常开发工作都在这个界面中进行。

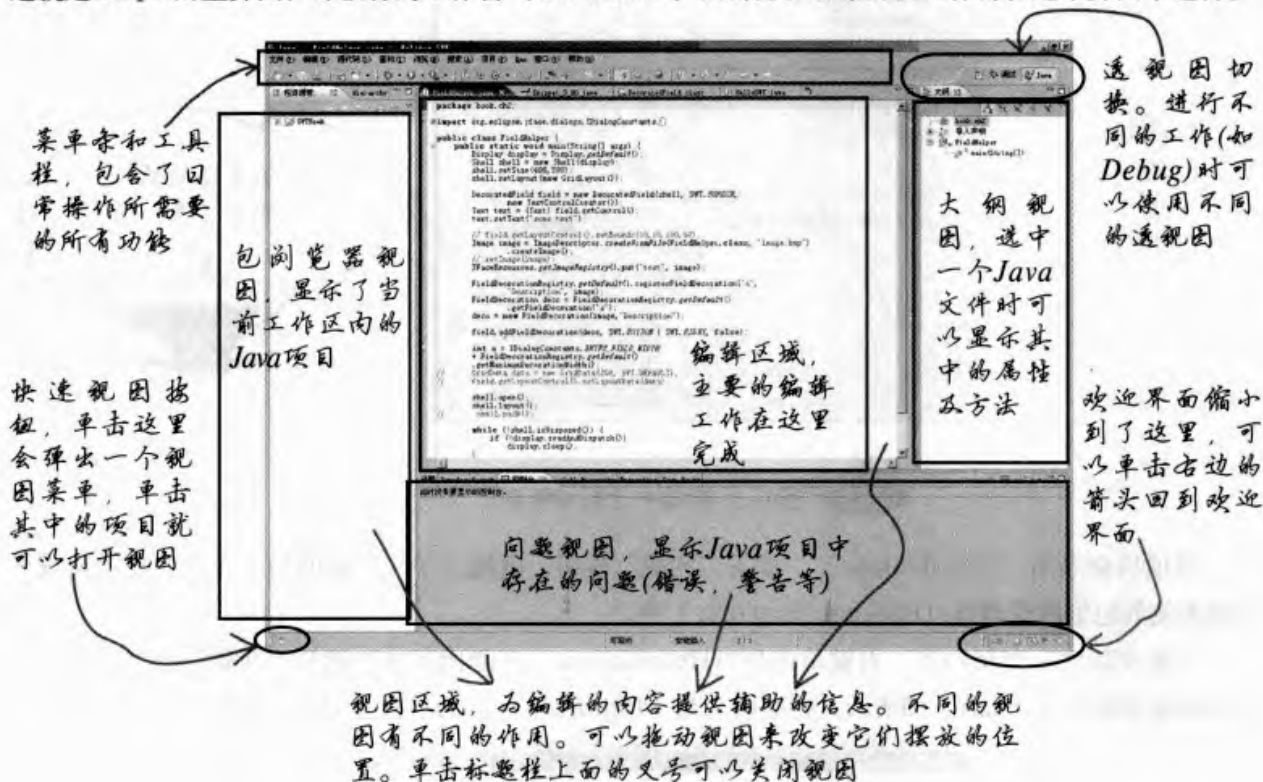


图1-3 Eclipse主界面

下面将开发一个Java程序“Hello Eclipse”以帮助读者熟悉Eclipse的使用。

首先，新建一个Java项目。项目是Eclipse用于存放和管理程序源代码的单元。单击菜单“文件”→“新建”→“项目”以打开“新建项目”向导对话框，如图1-4所示。选择“Java”→“Java项目”并单击“下一步”按钮。



图1-4 “新建项目”向导对话框

在下一步的向导页面中，可以为项目命名、选择存储目录（默认会存储在工作空间中以项目命名的文件夹中），以及其他设定等，如图1-5所示。

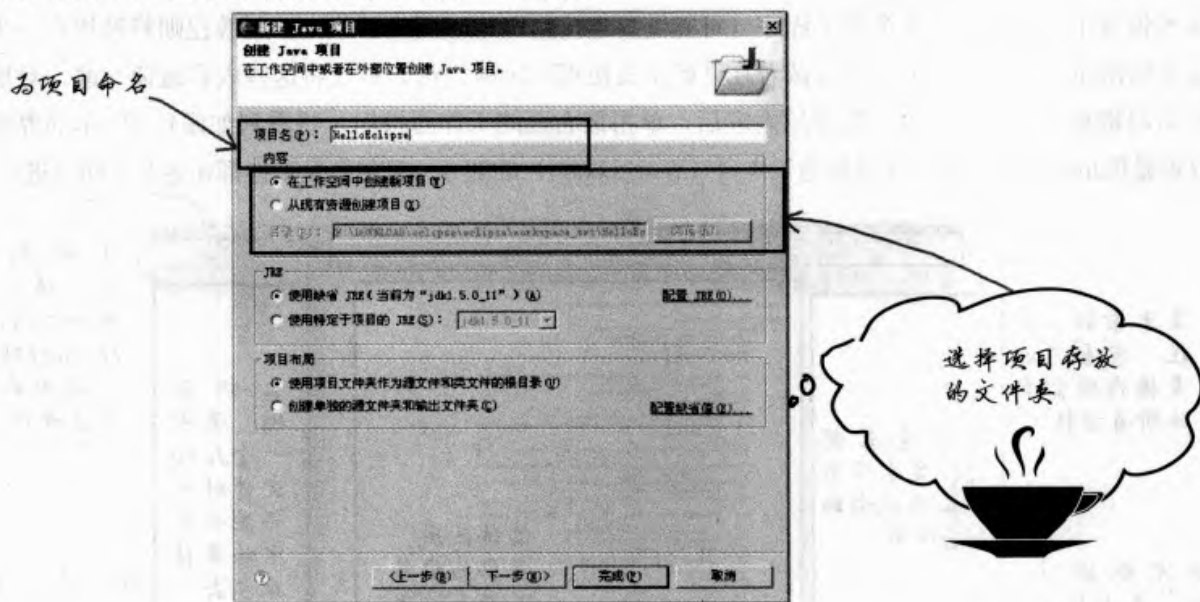
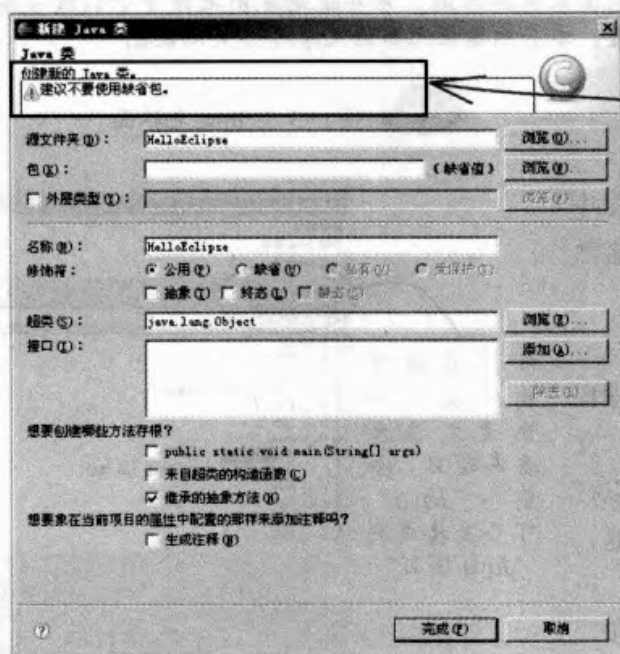


图1-5 新建一个Java项目

将项目命名为“HelloEclipse”，单击“完成”按钮，就建立了一个新的Java项目。它会出现在屏幕左侧的包资源管理器（Package Explorer）中。

下面来新建一个Java类。右键单击项目HelloEclipse，在弹出菜单中选择“新建”→“类”，打开Java类创建向导，如图1-6所示。将类命名为“HelloEclipse”后单击“完成”按钮。



如果输入的内容有问题，这里会显示错误信息或警告信息



图1-6 新建一个Java Class

新建的Java类会出现在左边的项目视图中，Eclipse会为它自动打开一个代码编辑器，如图1-7所示。这个编辑器具有关键字自动加粗变色、排版等多种功能。

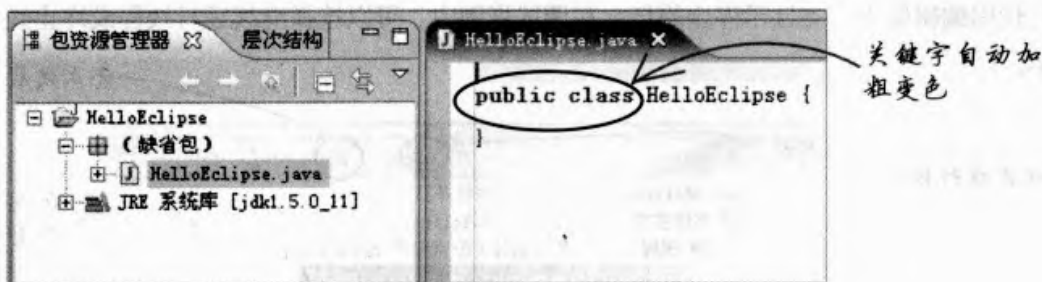


图1-7 Java代码编辑器

Eclipse为命令行程序提供了一个控制台视图（Console），在里面会显示用System.out输出的内容，如图1-8所示。在菜单栏中选择“窗口”→“显示视图”→“控制台”来打开控制台视图。



图1-8 控制台视图

在HelloEclipse类中编写一个main函数并加入一行打印语句“System.out.println(“Hello Eclipse”)”。然后在包浏览器中用右键单击这个类，在弹出菜单中选择“运行方式”→“Java应用程序”，在控制台视图中就可以查看到程序运行的结果，如图1-8所示。

使用Eclipse IDE还可以很方便地对Java程序进行调试。在编辑器窗口的左侧灰色条状区域处单击右键，在弹出菜单中选择“切换断点”就可以为这一行添加或取消断点，如图1-9所示。程序在调试方式下运行时，遇到有断点的行就会暂停。这时开发者可以查看变量取值、函数调用栈的内容等信息，以帮助查找问题。

选择“切换断点”可以添加或删除断点。如果只是暂时禁用断点，可以选择“禁用断点”。这时断点会变成灰色，不会发挥作用。

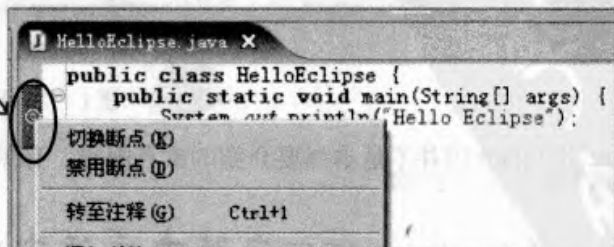


图1-9 设置断点

右键单击需要调试的类，在弹出菜单中选择“调试方式”→“Java应用程序”。IDE会切换到调试透视图（Debug Perspective）并以调试方式执行程序。当程序运行到断点位置时，会自动暂停，并在代码编辑器中高亮显示停止的行。在调试视图中，可以选择继续运行线程或终止线程，如图1-10所示。



图1-10 调试视图

调试透视图的界面如图1-11所示。

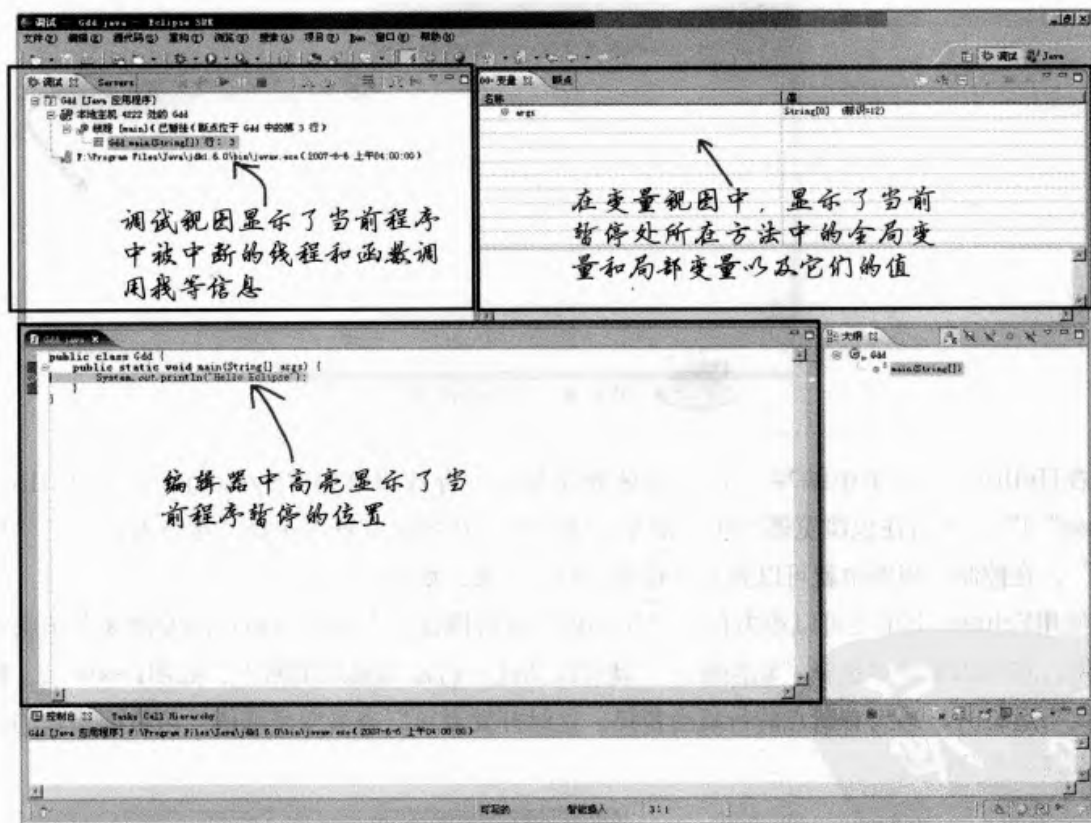


图1-11 调试透视图

Eclipse IDE的使用并不是本书要介绍的重点内容，有需要的读者可以通过其他相关书籍来学习。

1.1.2 为Eclipse IDE安装中文语言包

在Eclipse的下载网页 (<http://download.eclipse.org/eclipse/downloads/>) 中，可以找到一

个语言包 (Language Pack) 的下载链接。根据所安装的Eclipse不同版本选择合适的链接, 单击链接进入下载页面后可以发现语言包由数个集合组成, 其中NLPack1中包含中文(简/繁)以及德语、法语、日语、意大利语等语言的支持, 找到“SDK Language Packs”的下载项并下载NLPack1。

将下载的压缩包打开, 其中有一个同样名为“eclipse”的目录, 关闭Eclipse IDE, 将压缩包中的内容解压到Eclipse的安装目录的根目录下(如果Eclipse安装在F:\eclipse, 那么就把压缩包解压到F:\), 然后重新启动IDE。Eclipse会根据操作系统的语言设置使用对应的语言包, 如果用户使用的是中文系统, 现在出现的就是中文的IDE了, 包括所有的帮助内容也都变成了中文。

语言设置也可以手动改变, 这需要在启动Eclipse时通过命令行参数指定。新建一个指向“eclipse.exe”的快捷方式, 右键单击图标打开它的“属性”选项卡, 在“目标”中“eclipse.exe”的后面加上参数“-nl en_US”, 保存后双击这个快捷方式, 就可以重新使用英文版的Eclipse。

1.1.3 IDE的环境设置

使用IDE时, 有时需要改变一些默认设置以使它更符合自己的使用习惯。单击菜单项“窗口”→“首选项”, 可以打开首选项设置 (Preference) 对话框, 这里包含着Eclipse的所有环境设置, 如图1-12所示。



图1-12 “首选项”对话框

“首选项”对话框左侧是一个树控件, 列出了选项的类别, 单击类别左侧的“+”号可以将其展开; 右侧是被选中的项目的具体设置内容。下面只对一些常用的设置加以说明, 读者可以在学习过程尝试改变其他设置。

1. 改变IDE界面字体

在“常规”→“外观”→“颜色和字体”选项中, 可以调整IDE的界面字体。其中包括了菜单、视图和编辑器的标题, 对话框以及Java编辑器的字体及背景色设置。

2. 设置热键

在“常规”→“键选项”中, 列出了所有Eclipse IDE的菜单项以及它们所对应的热键。可以改变默认的热键设置或为自己常用的菜单操作添加热键。

3. 同时使用多个版本的JRE

在开发过程中,可能需要使用不同版本的JRE。在“Java”→“已安装的JRE”选项卡中,可以看到现在已经安装的JRE。单击“添加”按钮可以添加一个新的JRE。添加完毕后,新的JRE会被加到已安装的JRE列表中,如图1-13所示。可以单击前面的复选框来确定当前要使用哪一个JRE。如果改变了默认的JRE,所有现在处于打开状态的Java项目都会被重新编译。

已经安装的JRE列表。前面的勾选代表这个JRE是默认的,在新建项目时会优先使用它

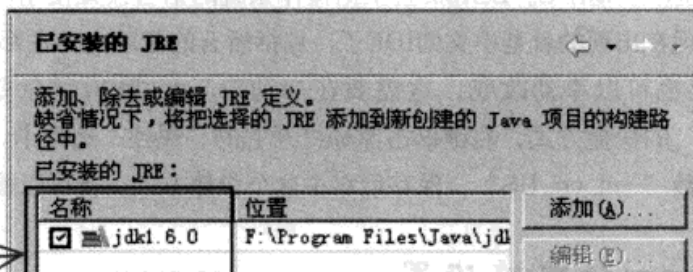


图1-13 已安装的JRE列表

这个选项是IDE级别的,用户只能指定当前IDE使用哪一个版本的JRE,而不能为每一个项目指定一个IDE版本。

4. 改变Java的编译选项

在开发过程中,Eclipse的Java编辑器会自动编译Java文件并将错误或警告信息显示在编辑器上,如图1-14所示。

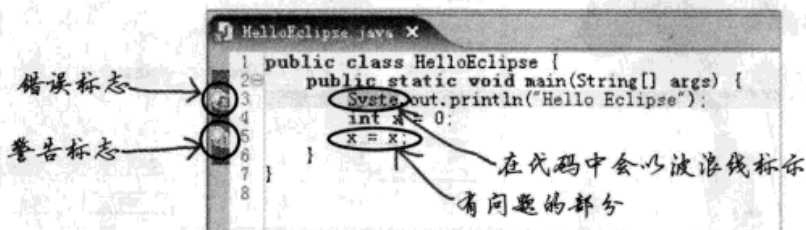
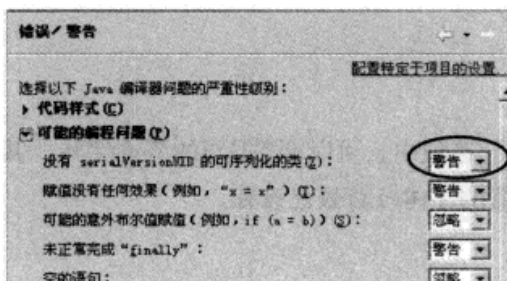


图1-14 编辑器中显示的错误信息

在“Java”→“编译器”→“错误/警告”选项卡中,可以决定应当把哪些问题显示成错误、哪些显示成警告、而哪些又不显示出来,如图1-15所示。



在下拉框中选择问题的严重级别(显示为错误,警告或不显示)



图1-15 调整问题的警告级别

这个功能包含了很多检查，包括编码规范，有隐患的代码写法等内容，应用恰当的话，可以在开发时就帮助开发者找出代码中潜藏着的问题。

5. Eclipse IDE的自动升级

Eclipse IDE中包含了自动升级功能，可以自动链接到Eclipse网站检查更新、下载并安装它们。在“安装/更新”→“自动更新”选项卡中可以启用自动升级选项，在此处还可以设置何时检查升级内容，发现更新后是否自动安装等内容。

1.1.4 使用帮助系统

单击菜单“帮助”→“帮助内容”可以打开帮助系统，如图1-16所示。

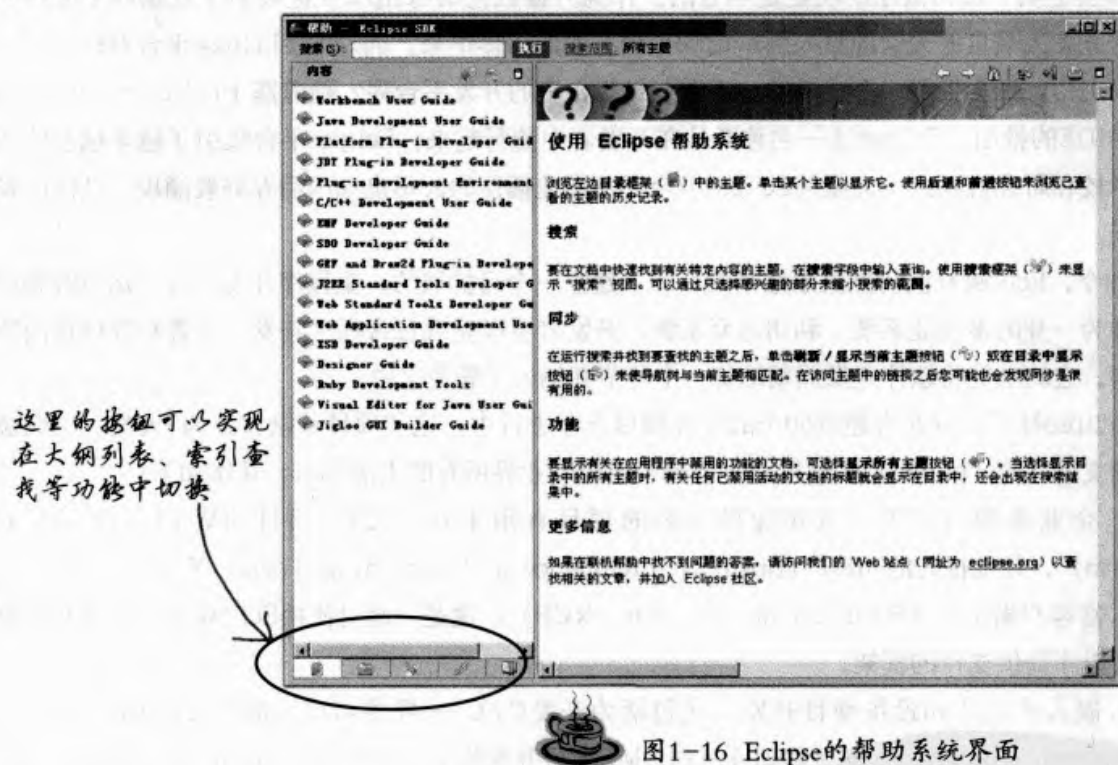


图1-16 Eclipse的帮助系统界面

可以单击左边的大纲列表打开帮助内容，也可以使用查询功能直接搜索关键字。Eclipse的帮助功能内容极为丰富，从工作台的使用到插件的开发都有详尽的介绍，除此之外还有一系列的示例代码可以参考。

1.2 什么是Eclipse

许多人提到Eclipse，总会将它和Java IDE等同起来。Java IDE的确是Eclipse平台下面最早期也是最成功的产品之一；然而，如今Eclipse所包含的内容，早已远远超出了Java IDE的范围。

Eclipse这个名称最早出现在2001年。2001年11月，IT业界几大巨头 Borland, IBM, Rational

Software, Red Hat, SuSE等公司共同创建了最早的eclipse.org网站, 它的管理者被称为监管委员会 (Board of Stewards)。当时网站的创建目的是利用Java语言开发一个用于软件开发的通用平台, 这一目的到现在仍未有变化; 然而当时的Eclipse社区是由大公司把持的非开放性平台, 这也导致了它发展缓慢。鉴于这种情况, 作为主要发起者和推动者的IBM开始思考别的出路。

2004年2月2日, 是一个值得纪念的日子。这一天, Eclipse社区监管委员会宣布, Eclipse社区正式独立成为一个非营利性的组织。IBM同时也宣布将其所拥有的Eclipse平台的源代码开放出来归社区所有。从那时开始, Eclipse社区 (www.eclipse.org) 作为一个独立的组织开始推动Eclipse平台的开发工作, 而这些开发的成果, 全部是开放源代码的。为保证Eclipse平台的充分健康发展, 社区监管委员会制定了Eclipse公共协议 (Eclipse Public License, EPL) 来管理平台的代码, 只要遵守EPL, 任何人都可以自由地使用这些代码而不需要付钱。

事实证明, IBM做出的决定是明智的。作为开源社区的Eclipse社区吸引了互联网上数以百万计的优秀开发人员自愿无偿地投入到Eclipse平台开发的工作中来。而EPL使Eclipse平台对软件公司同样具有吸引力。谁会拒绝一个成熟的, 不需要付版权费的开发平台呢? 随着基于Eclipse平台的支持各种语言的IDE的推出, Eclipse这一名称开始在开发者中流行起来。Eclipse平台吸引了越来越多的商业公司, 学校和研究机构。一大批为Eclipse平台开发商业插件的公司也如雨后春笋般涌现, 社区的发展如日中天。

如今, Eclipse社区仍然秉承最早的目的: 建立一个可扩展的, 集程序开发平台、运行时和应用程序框架为一身的多功能系统。利用这套系统, 开发者可以完成包含程序开发、部署和管理在内的一系列工作。它的功能将软件开发周期的每一个环节都纳入了管理之中。

Eclipse社区内现在有超过60个的开源项目正在进行中。这些项目可以被分为7大类, 它们被称为社区的支柱 (the pillars), 这里几乎包含了目前IT业界所有的主流技术, 具体如下:

1. 企业级项目开发。其中比较著名的项目有用于开发J2EE项目的WTP (Web Tools Platform), 开发报表的BIRT (Business Intelligence and Reporting Tools) 等。

2. 富客户端平台 (Rich Client Platform, RCP)。这是一套对使用Eclipse平台技术开发跨平台的桌面程序提供支持的框架。

3. 嵌入式系统和设备项目开发。这包括为开发C/C++程序而设计的开发环境 CDT (C++ Development Toolkit), 支持移动设备和J2ME项目开发的MTJ (Mobile Tools for Java), 以及用于嵌入式设备的富客户端开发平台ERCP (Embedded Rich Client Platform)。

4. 富Internet应用程序 (Rich Internet Application)。富Internet应用程序是指利用Ajax结合JavaScript开发具有丰富用户体验的Internet应用程序。和传统Web应用程序相比, 这一类程序的界面更加美观, 在用户体验上也更接近传统的桌面应用程序; 然而它又保留了Web应用程序轻量级的优点, 是程序开发的一个新的发展方向。ATF (Ajax Toolkit Framework) 致力于基于Ajax的项目开发。

5. 程序开发框架。这里除了支持各种语言的IDE外, 还包含了很多针对特定程序类型设计的开发框架。比如说致力于创建对象模型的Eclipse Modeling Framework (EMF), 专注于开发图形化编辑界面的Graphical Editing Framework (GEF) 等。

6. 程序生命期管理 (Application Lifecycle Management, ALM)。这包含了专注于程序性能

评估和测试的TPTP (Test and Performance Tools Platform)，用于控制Java程序开发流程的JWT (Java Workflow Tooling) 等。

7.面向服务架构 (Service Oriented Architecture, SOA)。STP (SOA Tools Platform) 项目是针对SOA理念设计的开发平台，这包含了从建模，流程设计到服务发布等一系列操作的支持，以及对BPEL (Business Process Execution Language，一种由IBM，BEA和Microsoft联合开发，用于描述基于Web Service工作流的语言) 的支持等。

Eclipse社区的新闻组(www.eclipse.org/newsgroups)和BugZilla数据库 (用于对Eclipse开发中的Bug进行管理的系统，网址为bugs.eclipse.org/bugs)都是对公众自由开放的，可以在Eclipse网站上找到它们的信息。开发者可以在这里寻找感兴趣的插件并下载，参与开源项目开发工作，也可以自己发起一个开源项目并号召世界各地的同伴共同参与进来。

下一节中，将开始对Eclipse平台的基本技术进行概要介绍。

1.3 SWT/JFace技术

一直以来，Java语言在企业级程序开发领域占据着主导地位，但在编写桌面应用程序方面却不尽如人意，人们甚至已经有了“Java写的GUI程序速度缓慢而且难看”这样先入为主的观念。但当Eclipse推出的时候，它精美的界面，让人耳目一新，也颠覆了上述成见。这一切都要归功于Eclipse底层的图形技术——SWT。

SWT (Standard Widget Toolkit, 标准图形工具箱) 是一种用Java开发GUI程序的技术。与出自Sun的AWT (Abstract Widget Toolkit, 抽象图形工具箱) 和Swing不同，SWT是Eclipse的开发人员自行建造的。它无论在界面美观度上还是在性能上都远远超越了AWT和Swing，即使和操作系统上的本地图形界面程序相比也毫不逊色，这都源自于SWT与众不同的设计理念。

对跨平台的支持，是Java语言的亮点之一，但对这一点却成了Java GUI技术设计最大的难点。不同操作系统的图形平台有着自己不同的显示风格，拥有一套不同的窗口控件。每种Java GUI都是这个难题的一个答案。AWT是由Sun提出的，Java最早的GUI技术，它采用了“交集”的办法来解决这一问题，即只支持每个图形系统都有的、最常用的那些控件和显示风格。这样做的好处是AWT中的所有控件都可以在本地图形系统中找到对应，可以像本地程序一样利用图形系统底层的消息处理机制来处理界面操作消息，对用户操作的反应速度快；其缺点就是支持的控件实在太少，使得页面单调难看，如今AWT技术已经几乎成为了明日黄花。Swing是Sun提出的另外一种GUI解决方案，它采用了和AWT完全不同的另外一套策略，图形平台除了自己一套标准的控件外，一般都会支持用户手绘界面的功能，Swing技术就是基于这一功能，将所有的控件都“画”了出来。这使得Swing可以完全控制界面的“长相”，也不再需要考虑目标平台是否支持特定的控件或显示风格，而保证了显示的一致性。但是有利必有弊，Swing完全脱离操作系统控件的支持，意味着它不能使用操作系统的消息处理机制而必须自己管理这些消息，这就使得Swing的性能相对来说受到了影响。幸好由于目前硬件发展的速度极快，一定程度上为Swing弥补了性能的缺憾。目前使用Swing开发的大型应用程序不多，典型的有Oracle公司的JDeveloper。

SWT技术是吸取了AWT/Swing的优点,综合而成的一种技术,它同时采用了AWT和Swing的一部分思想。SWT吸取各个图形平台的经验,决定自己的一个控件集合,然后针对某个目标平台进行判断,目标平台上有的控件,SWT会直接使用,以达到较快的处理速度和本地化的显示效果,目标平台上没有的控件,SWT则会采用Swing的方法进行绘制,“帮助”目标平台支持这个控件。如图1-17所示的是一个基于SWT绘制的窗口分别在Windows, Linux和MacOS X下运行的显示界面。



不加修改的同一个SWT程序,在不同图形平台上运行时,就会带有不同目标平台的特征。可以注意到图中Windows平台, Linux GTK平台和MacOS X平台对按钮、滚动条等控件的不同实现



图1-17 SWT绘制的窗口

JFace则是一套基于SWT的工具箱。它将一些常用的界面操作包装了起来,对界面设计进行了更高层次的抽象,使得开发人员可以抽出更多精力关注程序的业务逻辑,而不是不断地编写重复的界面控制代码。JFace的设计目的是和SWT协同工作,而不是将SWT的实现隐藏起来。开发者完全可以在程序中同时使用JFace和SWT。在Eclipse平台中,广泛使用了JFace。

JFace为应用程序中的一些常见功能,如“对话框”、“首选项设置(Preference)”和“创建向导(Wizard)”等,提供了实现框架,大大方便了开发者。此外,它还有两个最吸引人的功能,操作(Action)机制和查看器(Viewer)机制。操作机制使用户可以将一系列和界面无关的操作,如对底层数据的修改等,封装成一个操作对象,再将这个对象和任意的界面控件关联起来,以便于重用这些操作代码;查看器则是对一些复杂控件,如表格(Table)、树(Tree)和列表(List)做的包装,使这些控件使用起来更加便捷。查看器使用了MVC模式,开发者只需要修改底层数据模型(表格和列表的底层数据模型通常是一个数组,而树的则是一个树形结构)的数据就可以改变控件上的显示。在最新的3.2版中,JFace提出了数据绑定(Data-binding),这可以视做对View的进一步扩展,允许用户将底层的数据模型和界面上的所有控件,如文本框等进行绑定,当数据模型的内容发生变化时,控件的内容可以即时变化;反之,当用户对控件的数据进行操作时,底层模型的数据也会同时被修改。

1.4 插件技术和OSGi



Eclipse平台包含有很多功能模块,如JDT(Java Development Toolkit)等模块的历史几乎和Eclipse一样悠久,以至于人们认为它们是Eclipse平台的不可或缺的组成部分,但事实并非如此。Eclipse平台不包含任何专用的模块,它的设计遵循微内核理念,即提供最基本的运行时环境、简单的系统服务和模块之间交互的渠道,而各式各样的模块则使用这些服务运行并相互交流。

为了处理模块之间的依赖关系，Eclipse提出了扩展（Extension）和扩展点（Extension Point）的概念。每一个希望被别的程序模块扩展的模块，都必须声明一系列的扩展点，可以将这个想象成插座；而希望在这个模块上扩展功能的程序模块，就需要按照扩展点的声明来编写扩展，这个就是和插座配套的插头了；扩展点提供服务功能，而扩展使用这些服务。Eclipse平台本身就声明了一系列的扩展点，供其他模块使用它提供的服务，在组成Eclipse平台的内部模块之间，其交互也同样遵循着扩展点和扩展的规则。

将平台和其他各种功能组件插接起来，就构成了一个可用的程序体系。比如平台加上JDT模块就构成了一个Java IDE，加上CDT 则是一个C/C++ IDE。凡遵循这套扩展规则的模块，都可以方便地往体系中添加或删除，即“可插拔”的，因而被称为“插件”（Plug-in），如图1-18所示。插件技术是Eclipse平台最重要的技术之一，也是平台高扩展性的由来。在Eclipse平台体系中，一切应用都是插件。

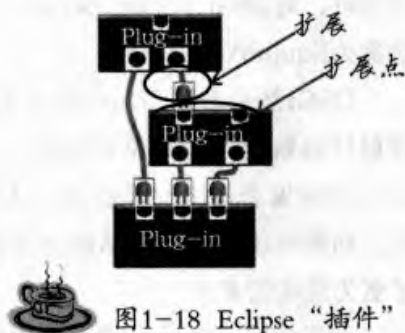


图1-18 Eclipse “插件”

插件技术使得系统的结构简练，可维护性与可扩展性大增。然而，一切都是插件，使得插件的数量同样大增。如果将这些插件在平台启动时全部装载到内存中，将对系统的性能造成极大影响；而且因为用户不可能同时使用所有这些插件，一次性全部装载进来也是没有必要的。为了解决这个问题，Eclipse采用了延迟装载（Lazy Loading）的技术，只有在一个插件被其他模块调用的时候，才将它装载到内存中。

通过将扩展的声明和实现分离，Eclipse完美地实现了延迟装载。在这套体系内，扩展和扩展点的声明，都是通过遵循特定规范的XML文件完成的。这些XML文件被称为“清单”（Manifest），在这些清单中指明了具体实现扩展功能的Java代码。清单描述了一个插件能够做什么，而代码则具体完成这些功能。这样，在系统启动时，只需要搜索所有的清单，并利用它们建立起一张索引表，就能够知道系统中有哪些插件以及它们能够提供什么样的服务，而不再需要一次性将所有的插件都读到内存里面了。这大大减轻了系统启动时的负载。使得“一切都是插件”的理念得以贯彻扩展的结构和实现，如图1-19所示。

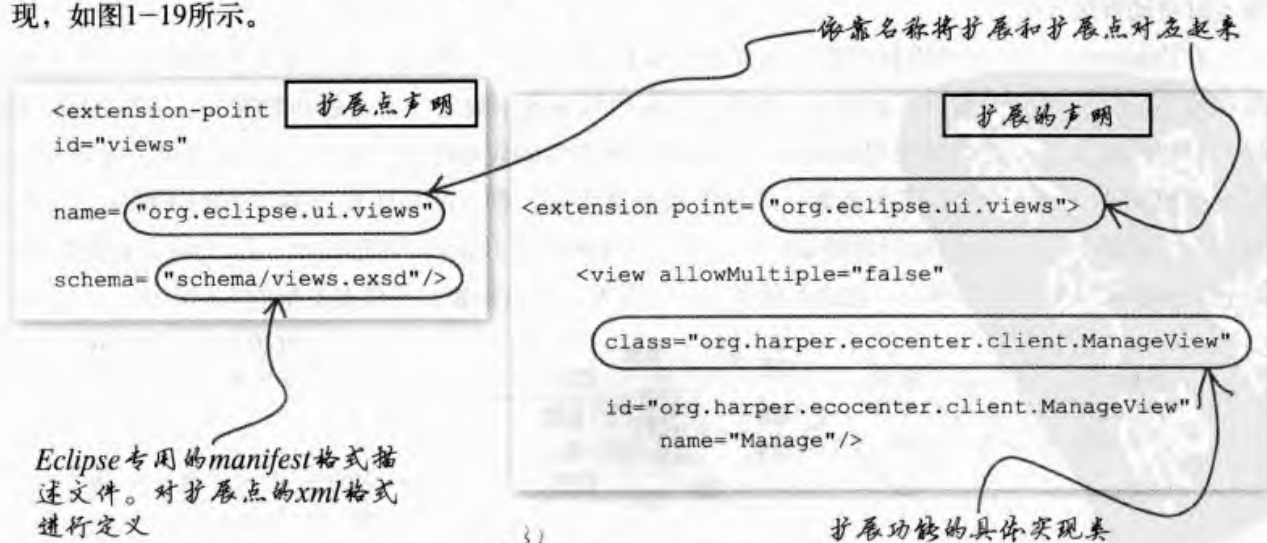


图1-19 扩展的结构和实现

Eclipse平台的插件技术的确构思精巧，然而最初的框架设计仍然有一些缺陷，这逐渐使得它的发展受到了限制。比如说，早期的Eclipse系统启动时会检查所有存在的插件，并构造一张静态的插件索引表，而这张表是不能在运行时修改的，这就使得每次添加或删除插件时，都必须重新启动整个平台。从3.0开始，为了使这套体系获得更大的灵活性和生命力，Eclipse的开发人员们对内核进行了重新构建，新的内核在保留原有声明与实现分离的插件技术（这套技术通常被称为Eclipse Runtime）的同时，对OSGi（Open Services Gateway initiative，开放式服务网关协议）做了实现，新的框架被称为Equinox。

OSGi是一套基于Java的开放式接口协议。它的思想与Eclipse插件的思想有些类似，都是希望将程序分解成小的可重用组件，并通过组合它们来构成功能更强大的可执行程序。然而OSGi对于Eclipse开发者最大的吸引力，还在于它的框架体系实现了在运行时对模块动态添加和删除功能的支持。新的Eclipse内核体系综合了传统Eclipse插件框架和OSGi框架的优点，为Eclipse平台的发展打下了更为坚实的基础。

关于OSGi的更多信息，可以访问其官方网站<http://www.osgi.org/>。

1.5 RCP技术

Eclipse RCP是帮助开发者创建和部署富客户端的平台。所谓富客户端，是指为用户提供了丰富功能体验的客户端程序。通常来说，一个富客户端程序应该具有以下特征。首先，应该是桌面程序而不能是Web应用程序；这也就意味着它可以暂时性地脱离网络运行，还可以对客户本地端的资源，如文件系统、打印机等进行操作。其次，界面美观，功能丰富；具有跨平台性并且可以和操作系统的图形平台紧密结合（能够支持拖曳操作，可以在系统托盘中显示等）；对用户的操作响应性好（在网络不畅时不能停止响应）。第三，富客户端可以承担一定程度的数据缓存以及业务逻辑计算功能，尽量减少和服务端端的交互，减轻服务器端的负载。通常，富客户端还需要包含一些辅助功能，如出现新版本时自动升级系统等。

在Eclipse平台的初始设计目的中，并不包含RCP的构想。早期，社区开发人员的目光主要聚焦在使用Eclipse平台开发IDE系统上；然而，社区中从来就不缺乏富于奇思妙想的人。从2.1版本开始，有的开发人员开始尝试使用Eclipse平台构建其他类型的应用程序。到了Eclipse 3.0，这个设计正式成熟起来，平台中和IDE关系紧密的插件被剥离出去，剩下的插件集合被作为RCP推出。这里面包含了新的基于OSGi的运行框架Equinox，基于SWT/JFace的图形模块，Eclipse平台的UI和Runtime模块。如图1-20所示。在这些插件之上，开发者可以构建几乎任意类型的应用程序。

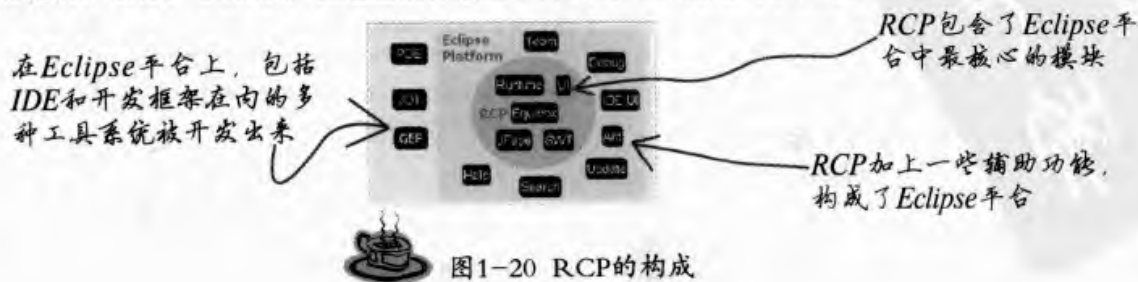


图1-21展示了IBM的产品Lotus Notes Hannover的运行界面。这是一个基于RCP开发，集电子邮件、工作流程管理等功能为一体的办公系统。可以看到，使用RCP开发的图形界面程序外观不逊色于任何基于其他开发平台的产品。

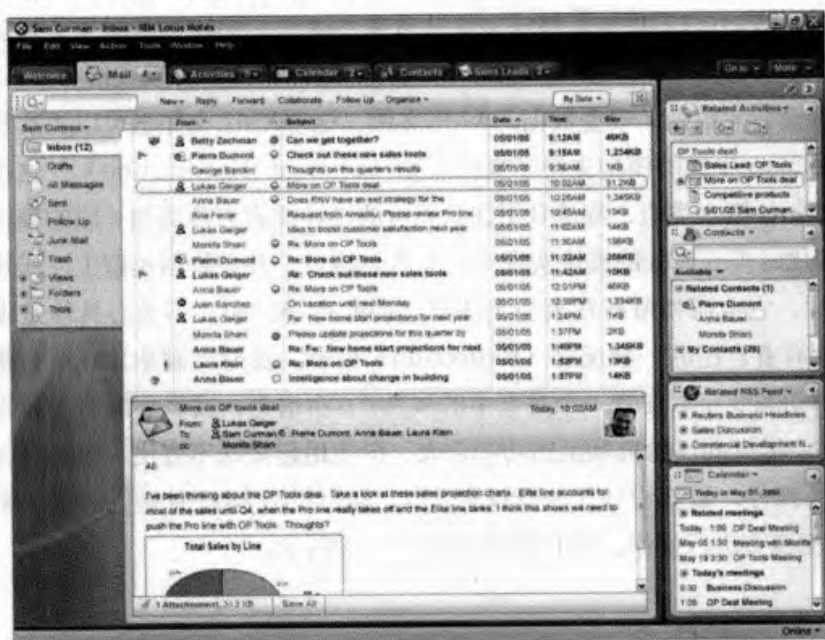


图1-21 Lotus Notes Hannover

RCP是构建一个基于Eclipse平台技术的应用程序所需要的最小集合，但这并不意味着基于这一技术构建的程序只能够使用这个最小集合中的插件。开发者可以随意选取其他插件并将其添加到这一集合中，以构建功能更为强大的应用程序。比如Eclipse平台中的UpdateManager可以随时升级已有的插件或添加新的插件，使得升级应用程序的过程变得简单可靠。而Help与Intro功能使得编写用户手册不再是繁杂困难的工作。

另外，开发者也可以在Eclipse社区中寻找各种开源的插件添加到自己的程序中，这些插件功能繁多，而且往往具有企业级应用的强度，妥善使用它们可以大大减少重复开发的工作。较著名的有用于报表设计与开发的BIRT等。

1.6 EMF技术



在编写面向对象的程序时，首先必须将实际生活中的例子抽象成Java对象来建立结构化的数据模型。EMF (Eclipse Modeling Framework, Eclipse建模框架) 就是一项致力于简化建模工作的项目。用户只需要描述需要建立的模型，就可以通过EMF生成健壮的、易于使用的数据模型实现代码。EMF项目由以下三大部分组成。

★EMF 这是EMF项目的核心框架。

EMF框架允许用户通过编写Java接口，从Rose导入UML类图等方法生成一个描述用户需要的数

据模型的元模型 (Meta Model)。在EMF项目中, 这个元模型被称为Ecore。它描述着数据模型中包含哪些对象, 对象中有哪些属性和方法, 以及这些对象之间的关系。用户可以直接从这个元模型生成数据模型的实现代码, 这些代码是基于EMF运行时框架的。EMF框架为数据模型提供了丰富的附加功能支持。首先, 用Ecore生成的代码可以方便地存储成文件并从文件读出。EMF运行时框架提供了对XMI (XML Metadata Interchange, XML元数据交换) 的支持, 这样, 用户可以直接将用Ecore生成的数据模型存储成同时包含模型元数据和模型数据的XML文件, 这些XML文件也可以再次被读出并建立起数据模型。其次, 用Ecore生成的模型支持了变化通知 (Change Notification)。这项技术和JavaBean中的Property Change概念类似, 都是利用Observer的设计模式来降低相关联的模块之间的耦合性, 希望在数据模型变化时得到通知的模块实现一个监听器接口, 并在数据模型上注册这个监听器, 当数据模型发生变化时, 它会通知所有在模型上注册过的监听器, 而并不关心具体是谁在监听。最后, 生成的数据模型拥有着自己的一套反射 (Reflection) 机制。反射机制是Java语言提供的, 它允许开发者在程序代码中动态地获取任意一个类型中的属性/方法信息, 或者获取对象属性的值, 调用对象方法的机制。Java中提供的反射机制虽然功能强大, 但使用起来既不方便, 安全性又难以保证。EMF针对数据模型的特殊性, 重新设计了一套反射API。通过EMF运行时提供的这一系列API, 用户可以安全方便地对数据模型进行操作。EMF的原理如图1-22所示。

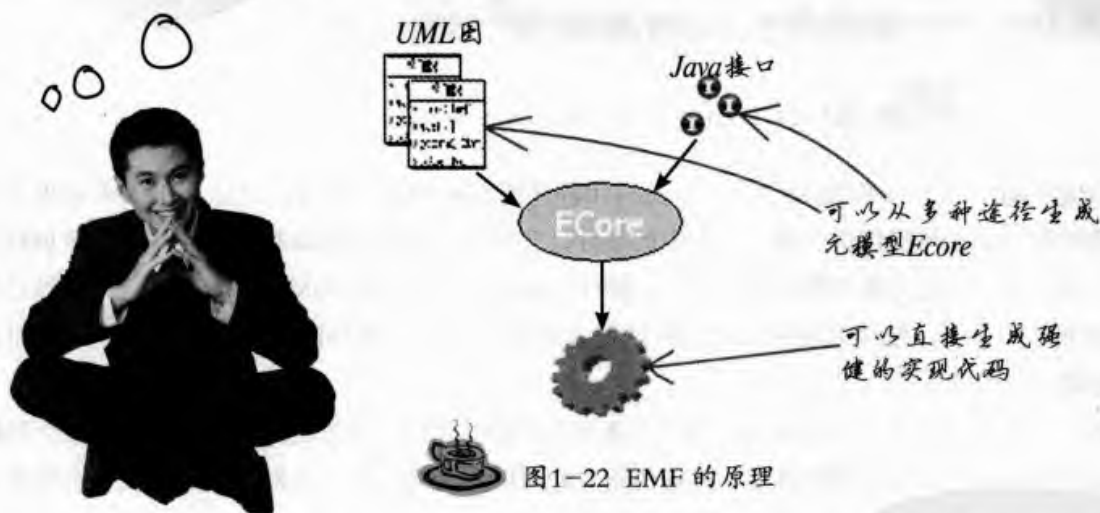


图1-22 EMF 的原理

★EMF.Edit 这一部分被称为Edit, 是因为它提供的功能都和数据模型编辑相关。它为数据模型提供各种功能的适配器, 使得生成的数据模型可以直接作为Jface的内容提供来源, 也可以在Eclipse的属性视图中显示出来并可编辑。这部分框架需要和CodeGen配合使用。

★EMF.CodeGen 从名字就可以看出, 这部分是用于代码生成的。前面提到, 用户可以直接从Ecore中生成数据模型的代码, 而具体的生成工作, 就是由这部分来完成的。生成的数据模型代码除包含数据模型的接口和实现外, 还包含了一个工厂类用于生成数据模型的实例, 以及一个Package类型, 其中包含了数据模型的元数据。同时, CodeGen也可以生成一个基于Eclipse RCP的编辑器, 用来对数据模型的内容进行编辑。

EMF虽然功能强大, 但是也有它的局限性。它主要适用于模型驱动 (Model Driven) 的程序, 也就是说, 要求能够从业务模型中抽象出相对稳定的底层模型。而对于企业级开发中所常见的业务驱

动 (Business Driven) 型程序, EMF就显得有些力不从心了。

1.7 GEF技术

图形化编辑可以直观地显示模型对象的属性和对象之间的关系, 在很多情况下都很有用, 如UML类图设计, 流程图设计等。GEF (Graphical Editing Framework) 是为了方便开发者开发基于RCP的, 支持图形化编辑的程序界面而设计的一套框架。GEF可以用来开发几乎任何和图形界面相关的应用程序, 如UML类图、流程图、GUI设计工具, 甚至是所见即所得的排版工具等。图1-23显示了使用GEF开发的一个逻辑电路模拟器。

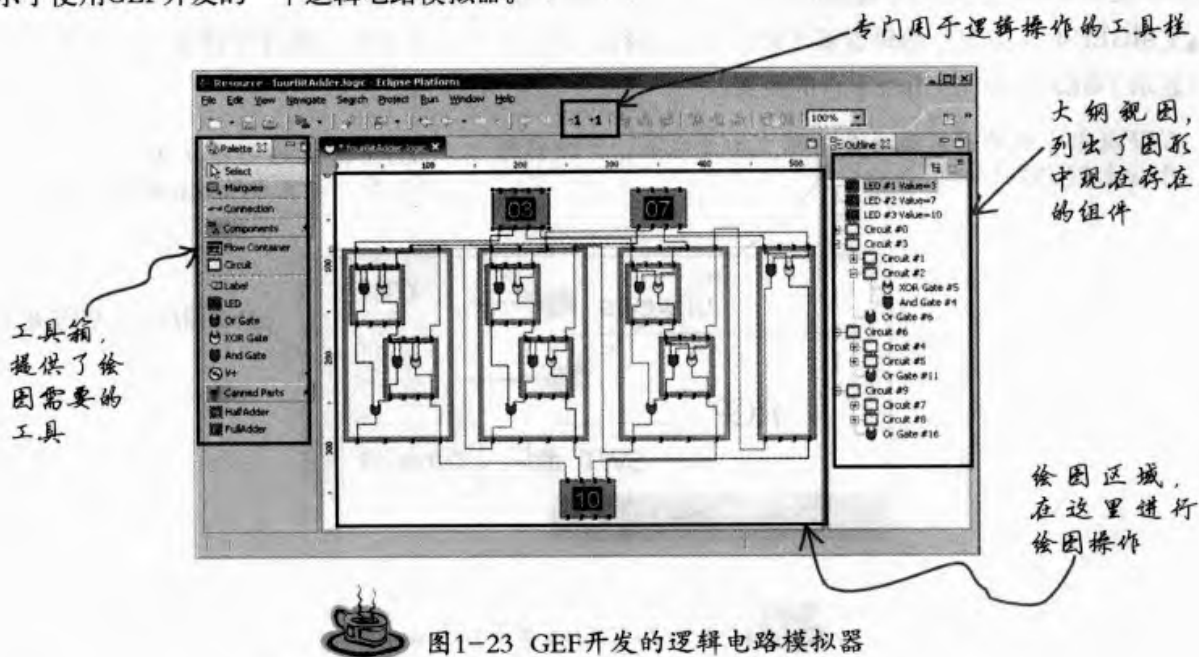


图1-23 GEF开发的逻辑电路模拟器

GEF的亮点有很大一部分在于它的图形绘制部分。和所有Eclipse项目一样, GEF的界面部分是基于SWT的。为了更加方便地绘制图形, GEF开发了一套基于SWT的轻量级绘图系统, 称为Draw2D。Draw2D的所有操作都基于一个SWT Canvas对象, 在Canvas上面利用画线、填充等基本操作进行画图。在第1.3节中曾经提到, Swing也是一套轻量级的图形系统。Draw2D之于SWT, 就相当于Swing之于AWT。Draw2D中的很多概念和设计思想, 都直接从Swing借用过来。然而, Draw2D和Swing的设计目的不同, 导致它们仍存在很大区别, Swing致力于GUI开发, 所以用Swing画出来的主要是文本框, 按钮等图形界面控件; Draw2D的设计目的是为了图形化的编辑, 因此Draw2D将精力更多地集中于这方面所用到的技术, 如图形的复合嵌套, 图形之间的连接线绘制等。如图1-24所示是一个使用Draw2D在窗口中绘制UML类图的例子。

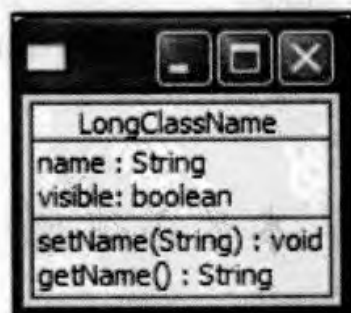


图1-24 Draw2D绘制的UML类图

GEF不仅能够将数据模型用图形的方式直观地展示出来,而且允许用户和模型进行交互(通过鼠标和键盘的操作添加/删除一个模型,修改模型中的文字等)。它的框架基于MVC的思想设计,将显示的图形与底层的数据模型分离开来,两者之间由控制器相连。

用户的操作动作被捕获后,由控制器翻译成针对数据模型进行操作的命令对象并执行这些命令;数据模型的内容改变后,控制器又收到了通知,随后会根据模型的变化刷新作为视图的图形界面。大部分的消息监听和转发工作都由GEF框架完成,用户只需要编写数据模型、Draw2D的视图以及少量的控制代码(如生成命令对象的部分)就能够实现对图形显示和编辑的功能。

GEF框架仅要求数据模型支持变化监听,其他代码可以向数据模型注册一个变化监听器,当模型内容变化时,会通过监听器通知这些监听者。只要模型满足这个条件,就可以用GEF为它生成图形化的表示。比如上节讲到的用于建模的EMF,就可以用来创建GEF可用的模型。

使用GEF生成的图形编辑器基于RCP平台运行,因此可以在任意RCP程序中使用这一技术。图1-25显示了GEF技术与Eclipse平台的关系。

GEF框架生成的图形编辑器依赖于RCP技术,可以使用于任意RCP程序中

GEF提供的一些工具(如画板等)都基于Eclipse视图开发

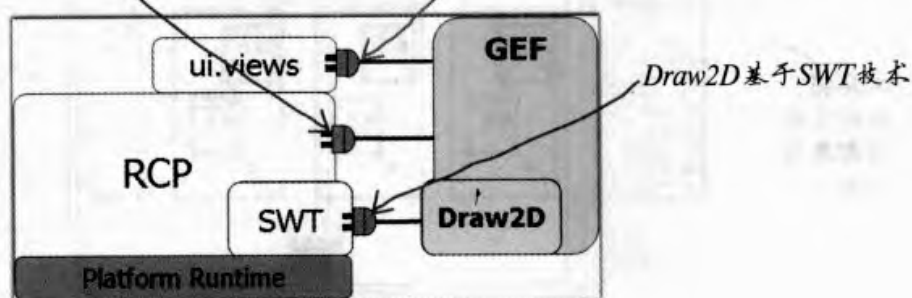


图1-25 GEF技术和Eclipse平台

使用GEF可以在很短的时间内开发出一个功能丰富的图形编辑器。在本书的第三部分中,将对使用GEF和Draw2D进行开发进行更详尽的介绍。

1.8 本章小结

本章对Eclipse平台及技术做了一个概览,简要介绍了SWT/JFace等技术的特点及其与Eclipse平台的关系。对Eclipse插件体系概念的了解,是学习Eclipse插件开发的基础,相信读完了这一章,读者对“什么是Eclipse”和“Eclipse插件开发能够做什么”的问题已经有了一个基本的了解。从下一章开始,本书将带您一步一步地进入Eclipse插件开发的世界。



第2章 SWT/JFace概述

拉开崭新的学习帷幕

上一章对整个Eclipse平台所包含的技术作了一个概述，而这些技术中最为基本的部分首推SWT。整个Eclipse平台的图形界面全部是基于SWT技术开发，因此掌握它是学习图形界面插件开发的基本要求；而JFace则是SWT的一个工具包，为SWT提供了很多方便的功能。本章将对SWT/JFace的体系架构作一个概述，并介绍如何在Eclipse IDE中编写、运行并发布一个SWT程序。通过本章的学习，读者将对SWT/JFace体系的基本架构有所了解，并为后面深入学习它们的使用打下基础。

本章内容包括：

- ★SWT结构浅析。
- ★SWT API结构。
- ★JFace。
- ★SWT与Swing。
- ★编写并发布SWT程序。



进入第02章

SWT技术是第一套基于Java的第三方图形工具库。它的设计思想是提供一套通用的API,使得开发出的图形程序不仅可以不加修改地在平台间移植,而且在外观上和速度上与使用C/C++等语言在操作系统平台上开发出来的本地图形程序毫无差别,还可以使用鼠标拖放操作、系统托盘等高级的系统服务。这些功能的实现,归功于它与众不同的体系架构。

要实现本地化的效果,程序一定要与本地图形操作系统发生交互,在与操作系统交互时,SWT使用了JNI技术。JNI (Java Native Interface) 是Sun公司为Java语言设计的用来与C/C++程序交互的技术。简单来说,可以将它理解成将Java语言编写的接口和C语言编写的函数绑定,从而使得调用Java接口就等于调用C函数的技术。使用JNI,也可以在C代码中反向操作Java代码的内容,其工作方式如图2-1所示。本书不对JNI做更深入的介绍,有兴趣的读者可以自行参考相关书籍。

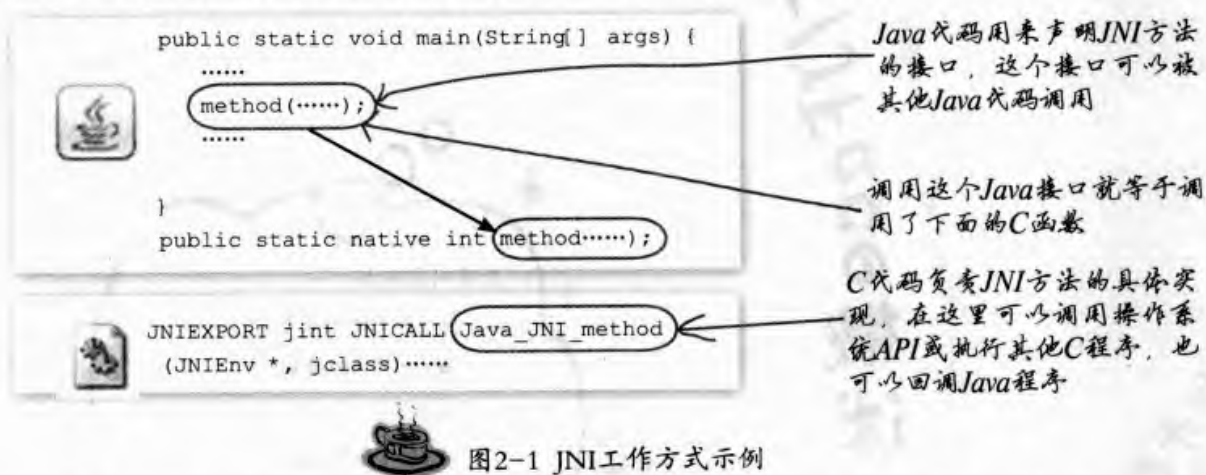


图2-1 JNI工作方式示例

目前主流的图形平台都提供了C语言的API供开发者使用。出于性能考虑,SWT并没有在操作系统的API上面再做封装,而是利用JNI将它们一对一地映射到了Java平台上。这些JNI函数的访问权限都被限制在了SWT的包内部,外部程序只能通过SWT的API使用它们,不能进行直接调用。针对不同的图形平台,JNI接口也各不相同,SWT API的任务就是封装这些不同点。这些API将基本的图形操作进行抽象,并将不同的JNI调用封装在内部,从而以一致的接口呈现给使用者。使用SWT开发,主要就是操作这些API。

2.1 SWT结构浅析

SWT的基本体系结构共分三层,如图2-2所示。

★第一层是SWT的API。外部开发者使用SWT时就是使用这些API编写SWT程序。这里包含了SWT的控件(文本框,按钮等)、事件处理(各种事件及监听器)等和图形界面开发者关系最为密切的部分;同时也只有这一部分的代码对外部开发者可见。底层C语言风格的API,在这一层上被封装为对象供外部开发者使用,但在封装的具体实现代码上,对于不同系统平台会有微小的差别。比如Win32平台的SWT在打开一个窗口时会针对WinCE和WindowsXP的特殊情况做不同的处理,而Linux GTK图形平台的SWT则没有这段代码。但从整体来看,这部分代码在不同平台上基本相似。

★第二层是JNI相关的代码。每个操作系统提供的API都拥有自己定义的一系列独特的数据类型

2.2.1 组件类

组件是用户界面的组成部分。在一个SWT程序中，界面上的所有内容，如按钮、文本框等，包括窗口本身都是组件。整个图形界面是由组件构成的。

SWT提供了数十种组件供开发者使用。了解有哪些组件可以使用，以及懂得它们各自适用在什么情况下，是学习SWT编程的基础。

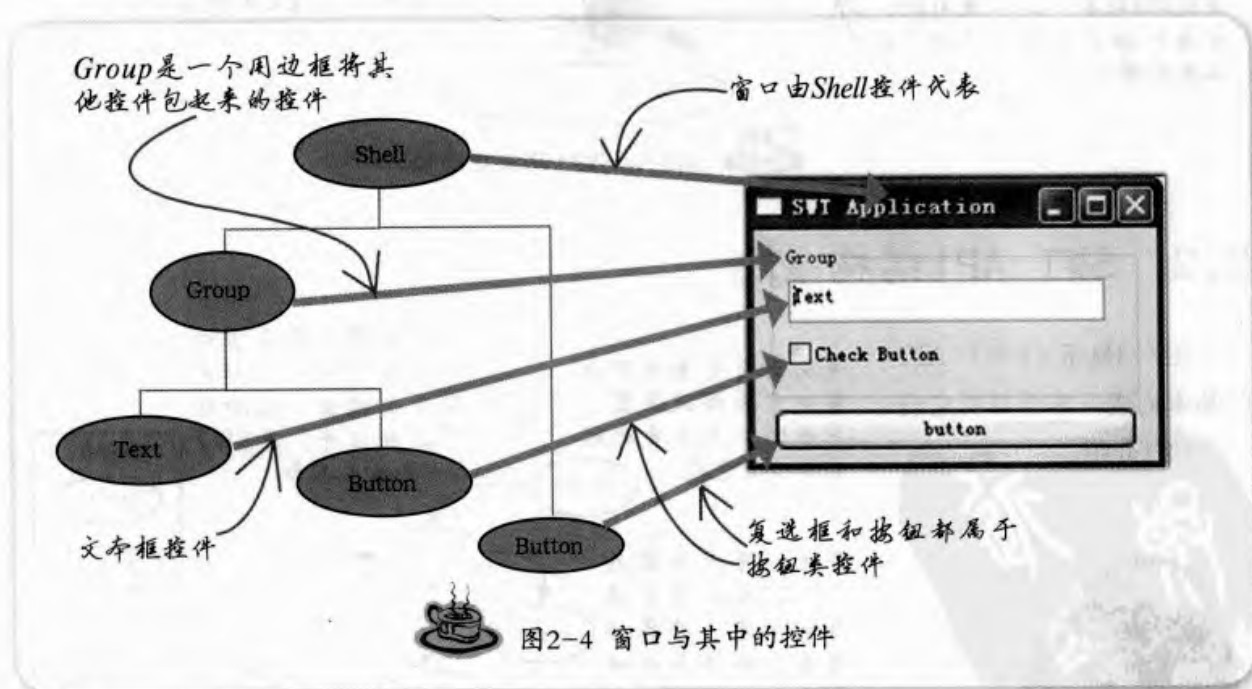
组件分为控件（Control）和项目（Item）两大部分。控件是界面的主体，项目依赖于控件并为它们提供附加的功能。标签控件可以用来显示文字；按钮可以让用户单击以启动一个功能；文本框控件可以显示文字，也可以让用户输入文字；单选按钮和复选框允许用户在提供的选项中做出选择；工具栏和菜单栏可以用来显示一些常用的操作选项以方便用户使用。关于这些控件的使用见本书的第4章。

对话框通常是比较简单的窗口，可以用来向用户显示一些提示或警告信息。在SWT中，还可以使用系统内置的文件对话框、颜色对话框、打印对话框等功能。关于对话框的使用也将在第4章中介绍。

有些控件是专门用来容纳、分组其他控件的，它们被称为容器控件。如窗口控件就属于容器控件。关于容器控件的使用见本书的第5章。

如果需要显示一系列的数据集合，可以使用列表（List）或表格（Table）；如果需要显示层次结构，可以使用树（Tree）控件。关于复杂控件的使用，将在本书第7章中介绍。

图2-4显示了一个窗口中所包含的控件以及它们的层级关系。



注意：

不能将控件类的继承关系和控件的相互包含关系混为一谈，两者之间没有必然联系。控件类的继承关系代表了控件的Java类型之间的关系，是固定不变的；而控件的包含关系则是这些Java类型的实例之间的关系，是和每个应用程序窗口设计有关的。属于子类的控件一样可以包含其父类的控件。

2.2.2 布局类

另外一类和页面显示关系紧密的部分是布局类 (Layout)，它主要是由布局管理器 (Layout Manager) 组成。当一个布局管理器被安装到窗口上之后，它就负责着在窗口中安排控件的位置和尺寸。当窗口尺寸变化时，布局管理器会根据一定的策略重新计算这些数据，并将控件安排在适应窗口新尺寸的位置上。不同的布局管理器，就代表了安放控件的一种不同策略。如图2-5所示为两个采用不同布局管理器的窗口在窗口尺寸变化时的不同表现。可以看到，窗口宽度缩小时，Demo1窗口中的按钮被摆放到了下面一排的位置；而Demo2窗口中的按钮宽度随着变小了。

布局管理器将窗口的布局设计与窗口中控件的具体业务逻辑分离开来，使得编程工作的模块划分更为清晰，也使得程序员可以根据需要动态地改变一个窗口中控件的摆放方式。

SWT中内置了许多布局管理器，在日常开发中，这些布局可以满足绝大多数需要。如果有特殊需求，还可以自己编写布局管理器。关于布局的详细介绍见本书的第5章。

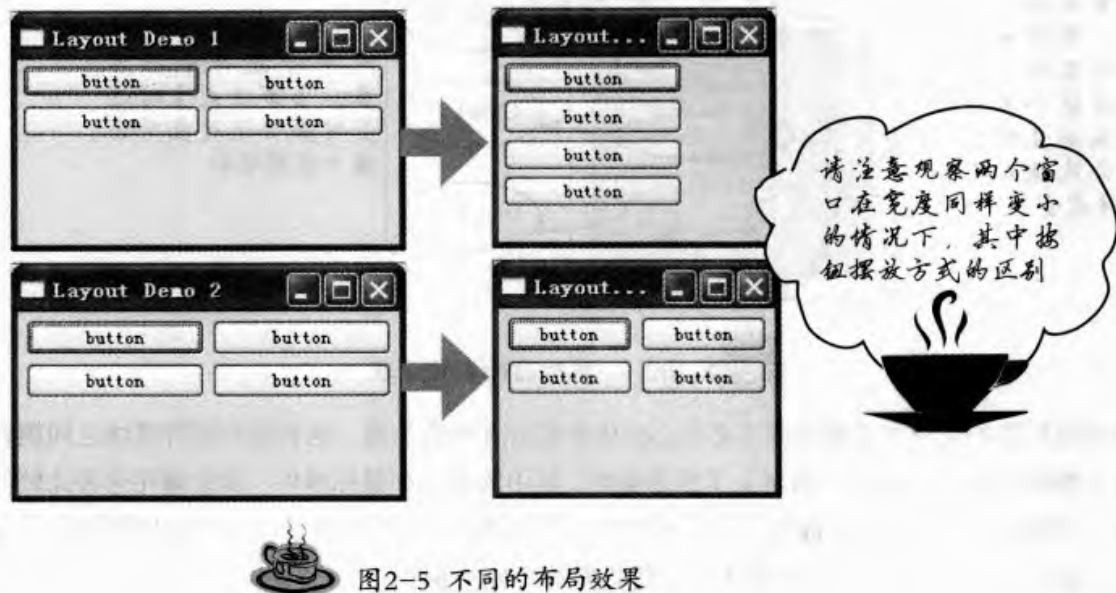


图2-5 不同的布局效果

2.2.3 事件类

事件类主要负责控件的消息传递。用户单击了界面上一个按钮，在文本框中输入一段文字，或者仅仅移动一下鼠标，都会产生一个事件 (Event)。事件是图形界面程序与用户交互的核心。SWT的事件处理机制采用了Java 1.1中提出的事件监听器 (Event Listener) 的结构，即使用Observer (观察者) 的设计模式来处理事件。对某个对象上的某个事件感兴趣的话，就需要在对象上注册一个对应事件的监听器。当事件发生时，所有在对象上注册过的此事件监听器都会收到通知。

监听器是一个接口，其中包含的每个方法都对应了某种事件的发生，而其参数则包含了所发生的事件的上下文信息。以鼠标事件MouseEvent为例，它对应的监听器是MouseListener，其中包括三个方法，如下面代码所示。从名字可以看出，三个方法分别对应了鼠标双击、鼠标按下和鼠标放开的事件，而其参数MouseEvent类，则包含了鼠标单击位置，哪一个按钮被按下等信息。


```
public void mouseDoubleClick(MouseEvent e);
public void mouseDown(MouseEvent e);
public void mouseUp(MouseEvent e);
```

每一个监听器方法都负责着一种不同类型的事件，从它们的名字就可以知道其监听的事件。相近类型的事件会被放在同一个监听器接口中。

如果希望为窗口中某一个按钮加上双击的处理函数，就要实现一个MouseListener接口，并在其mouseDoubleClick方法中加上处理代码，然后调用Button类的addMouseListener方法，将这个监听器添加到按钮的监听器队列中。这样当鼠标双击按钮时，mouseDoubleClick方法就会被调用。这里的按钮被称为事件源（Event Source）。如图2-6所示为监听器的工作原理。

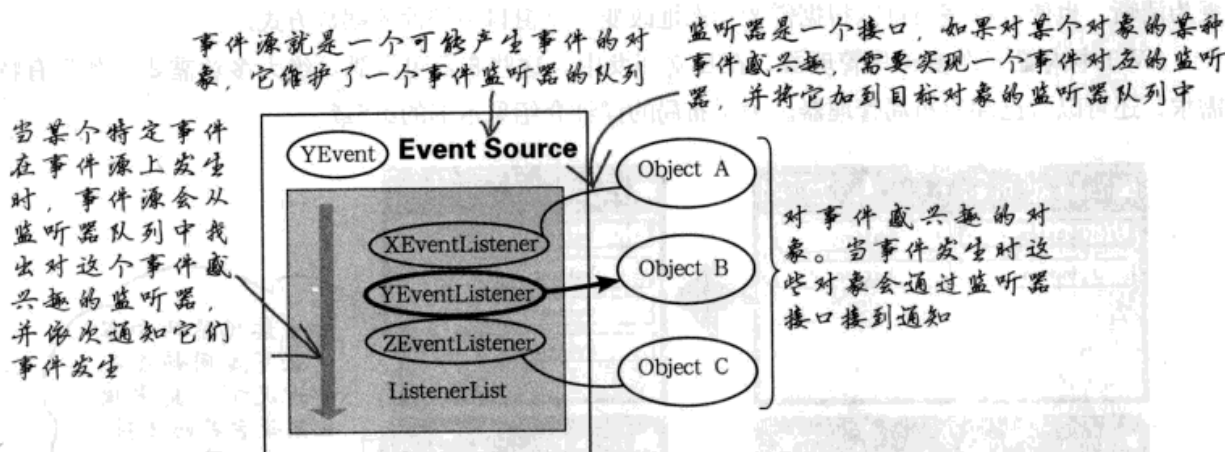


图2-6 监听器的工作原理

使用监听器的优点在于事件源不必关心具体的监听者究竟是谁。这种结构使得模块之间耦合度较低，易于维护和修改。SWT中预定义了许多事件，其中既包含如鼠标事件、键盘事件这种比较低层次的事件，又包括了文本框内容改变、复选框被选中等较为高层次的事件。开发者也同样可以定义自己的事件。表2-1列出了SWT中比较基本的一些监听器及对应事件。

表 2-1 SWT主要事件一览表

监听器	事件名	描述
MouseListener	MouseEvent	监听鼠标按钮按下的事件
MouseMoveListener	MouseEvent	监听鼠标移动事件
MouseTrackListener	MouseEvent	监听鼠标进入、离开事件源的事件
KeyListener	KeyEvent	监听按键事件
ControlListener	ControlEvent	监听控件尺寸或位置改变的事件
DisposeListener	DisposeEvent	监听控件被销毁（Dispose）的事件
FocusListener	FocusEvent	监听控件得到焦点的事件
SelectionListener	SelectionEvent	监听控件被选中（按钮被单击，复选框被勾选等）的事件

关于SWT事件处理方法见本书的第8章。

2.2.4 图形类与系统资源管理

图形类隐藏在SWT API的最下层，一般的开发人员很少会直接使用到它的内容。但了解这部分知识是深入理解和学习SWT的基础。图形类主要由两部分构成，即Resource和Display。

在编写图形界面程序时，所有控件、图片、字体等图形对象都会耗费系统资源。因为系统上可用的资源总是有限的，为保证这些资源被最大限度地利用，程序必须负责及时释放这些资源。资源在SWT中由Resource类的子类，如Font、Image等所使用，而分配或释放它们的操作则由Display负责和操作系统交互而完成。Display将不同的操作系统中对事件循环的实现封装成为SWT的统一事件循环机制，它还提供了丰富的方法用来访问操作系统的资源，因此可以将其视为SWT与底层操作系统交流的桥梁。

在SWT中，创建Display实例的线程被称为界面线程（UI Thread）。界面线程具有以下特性，或者说限制。

- ★Display的事件循环必须在它的界面线程中执行。
- ★所有通过Display完成的绘图操作都只能在它的界面线程中执行。
- ★在同一个界面线程中，不能同时创建两个有效的Display实例。

为了理解这些限制，有必要理解GUI图形界面程序运行的基本原理。GUI程序运行的机制就是针对用户在界面上的每一个操作事件做出反应，即所谓“事件驱动”。操作系统为每一个GUI程序分配了一个事件队列，用于存放属于它的事件。每当用户操作鼠标或键盘时，操作系统会根据鼠标单击的位置、当前的焦点窗口等信息决定应该把事件放到哪个队列中。而在GUI程序中，需要用一个循环不停地去读取自己的事件队列，每当检查到有新的事件时，就处理这个事件并做出适当的绘图动作反馈给用户。这就是事件循环（Event Loop）。这个循环不停地将消息传递到GUI程序的各个部分，好像一个水泵一样，因此也被称为“消息泵”。事件循环机制是GUI程序的心脏，GUI事件处理基本原理如图2-7所示。

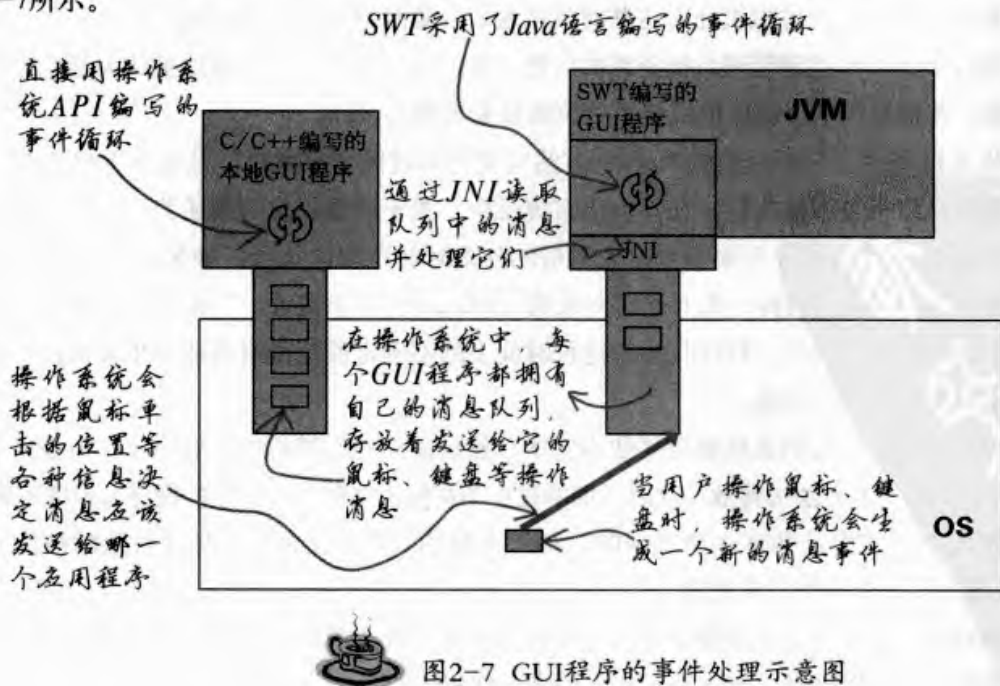


图2-7 GUI程序的事件处理示意图

在目前的主流操作系统中，对事件循环的处理存在着两种模型。一种是单线程模型，这种模型只允许程序中的一个线程监听事件消息并做出绘图动作；另外一种则是允许程序中的所有线程自由访问程序事件队列的多线程模型。为了保持平台兼容性，降低实现的复杂度，SWT使用单线程模型管理事件循环和绘图操作。这也正是存在上面三条限制的主要原因。

采用单线程模型的好处主要是，首先，使用单线程模型不仅能够更好地管理内存，减少操作系统中的线程数，也能促使程序将绘图操作集中在一起，实现业务逻辑与绘图代码的分离，结构更清晰，调试起来更容易；其次，在许多主流操作系统中，图形操作并不是简单地依次执行，专门的图形引擎会对这些操作进行优化以提高效率。将绘图操作集中在同一个线程中提交，有利于优化引擎最大程度地发挥作用。

※ 注意：※

试图在非UI线程中对图形界面进行操作不会导致编译错误，但却会在运行时得到一个“`ERROR_THREAD_INVALID_ACCESS`”的SWT异常。



在SWT中，没有系统资源的自动分配与释放机制，资源的管理是由程序员手动完成的。对于资源的使用与释放，有如下两条原则。

★谁分配谁释放 (If you created it, you dispose it)

★释放父资源时，所有子资源会被一起释放 (Disposing the parent disposes the children)

对许多从C++转行到Java的程序员来说，自己动手管理系统资源是一个很遥远的事情了。Java不是有垃圾收集 (Garbage Collection, GC) 机制么？为什么不能将这些资源的释放写在Resource类的finalize函数中，并由Java虚拟机在对这些对象做垃圾收集时释放系统资源呢？

关于使用finalization机制管理系统资源的问题由来已久。通常，这种做法被认为是危险的，而且容易滋生错误。其根源在于Java虚拟机规范对垃圾收集机制的设计。

在垃圾收集机制中，一个对象的finalize方法究竟什么时候会被执行，是完全不确定的，甚至有可能到Java虚拟机结束也不被执行。在Java语言规范中，关于垃圾收集的描述为：“Java编程语言并不强制规定finalize动作应该在对象被判定无效后的什么时间范围内执行。如果执行finalize方法时出现了异常，这个异常将被忽略掉，而对这个对象的finalize动作将会终止。”而对图形界面编程而言，系统资源是十分关键的。因此，垃圾收集的这种时间上的不确定性使得它不适合用来执行系统资源的释放，它可能导致以下几个问题。

★一个程序申请了大量的系统资源并使用完毕，然后放任它们等待被垃圾回收；但是由于垃圾收集还没有执行，这些资源无法被释放。这时，旧的程序不再使用这些资源，新的程序又申请不到它们。

★某些资源在系统中是共享存储空间的，如果大量的A资源被申请，却没有及时被作为垃圾收集，有可能导致申请B资源的行为失败。

如果使用垃圾收集管理系统资源回收，会使得程序对系统资源的使用方式过度依赖于具体的Java虚拟机对垃圾收集的实现机制，而这是不利于程序可移植性的。

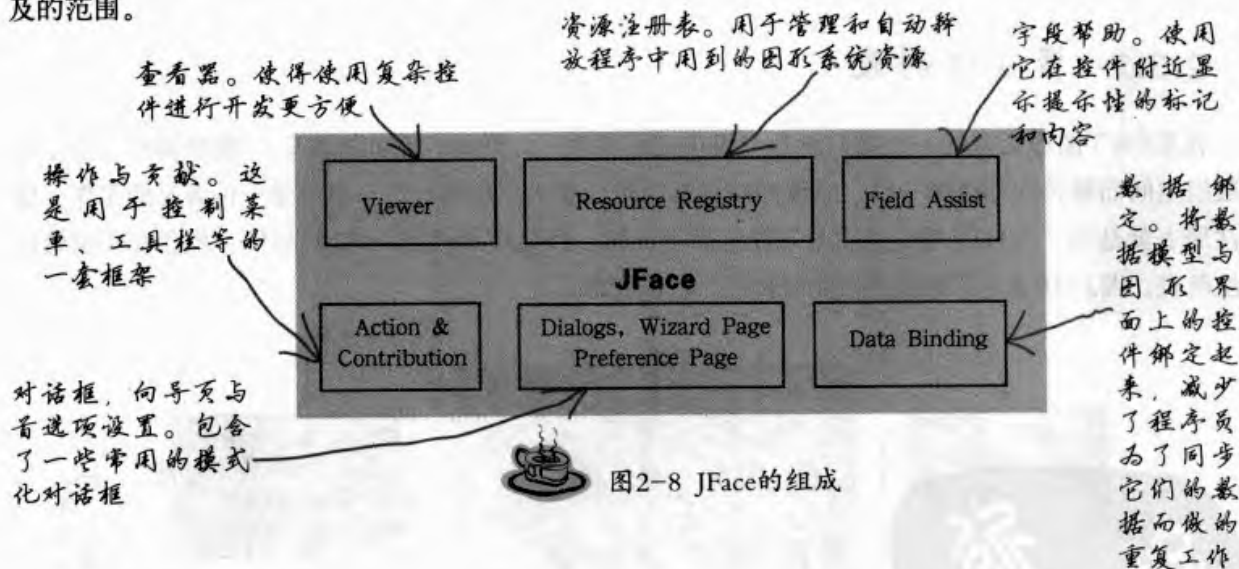
2.2.5 其他内容

SWT中还包含许多其他高级内容，如打印、拖放操作、系统托盘使用、OpenGL支持等。其中部分内容将在后面的章节中有所提及。

2.3 JFace

JFace是基于SWT的一套图形工具包，它没有为SWT提供任何新的功能，只是将一些较烦琐而且常用的图形操作封装了起来，使得开发工作更简便。JFace完全使用SWT API进行开发，并没有涉及任何SWT中平台相关的部分，所以JFace没有不同平台版本之分。

JFace的雏形是在Eclipse的开发人员编写IDE的图形界面时，为了方便而针对一些常用的图形界面开发模式（如对话框等）开发出来的一系列组件。通常用JFace写四五行代码就可以做到的事情，SWT需要写几十行甚至百行以上。但JFace并不能完全替代SWT，因为它只包含了少数几个最常用的模式。通常的工作方式是使用SWT完成大部分工作，并在适当的时候使用JFace。为了明确什么是“适当的时候”，就需要了解JFace究竟是对哪些常用操作做了封装。如图2-8所示列出了JFace所涉及的范围。



2.3.1 查看器

查看器（Viewer）是使用MVC模式对一些复杂控件做的封装，以方便开发人员使用这些控件。目前的查看器包括列表查看器（ListViewer）、表格查看器（TableViewer）、树查看器（TreeViewer）等。

查看器将数据模型从复杂控件中抽象出来，本身作为一个控制器来监听模型的变化和控件的变化，并负责传递这些变化。开发者不需要具体操作那些复杂控件，而只需要对数据模型进行操作，就可以改变控件显示的内容。当控件的内容在页面上被修改时，查看器收到通知后也会发给数据模型，数据模型接到通知后可以检查这些修改并决定是否接受。查看器并没有将控件隐藏起来，开发者可以

随时通过查看器提供的访问方法（如TableViewer的getTable方法，TreeViewer的getTree方法等）得到对应的底层控件对象并直接操纵它。表格查看器工作示意图如图2-9所示。

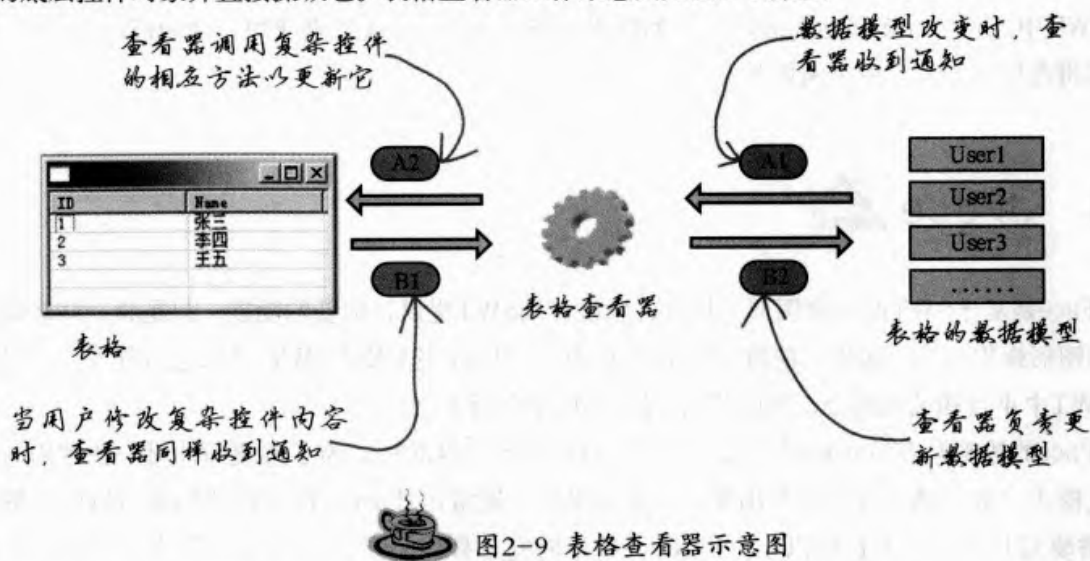


图2-9 表格查看器示意图

查看器是JFace中十分重要的内容。所有用到复杂控件的地方，基本上都可以而且应该使用查看器。关于查看器的使用见本书第7章。

2.3.2 资源注册表

使用SWT图形资源的一个原则就是“谁申请谁释放”，然而，正如许多C++程序员经常忘记释放数组空间而导致内存泄漏一样，当程序结构复杂时，要求严格执行这一点不是一件容易的工作。资源注册表就是为了方便管理常用的图形资源而设计的，目前JFace包含了针对字体、颜色和图像资源的注册表。图2-10显示了如何使用注册表进行资源管理。

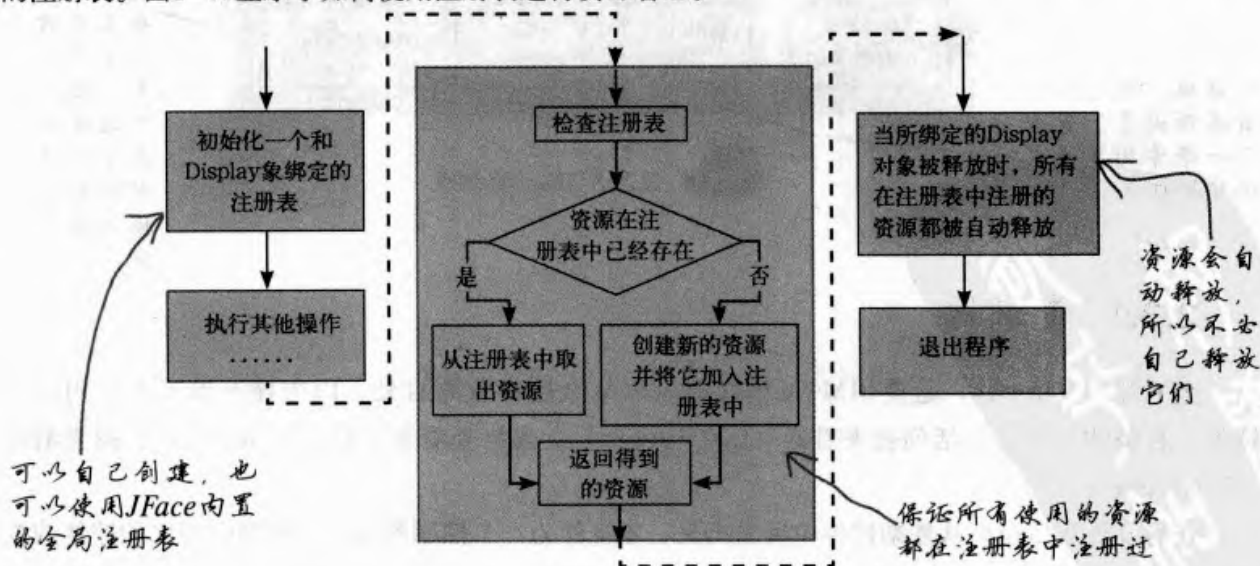


图2-10 资源注册表的使用

Face在JFaceResources类中为这些注册表提供了采用Singleton模式的实现。使用这些资源注册表，就可以有效地避免因为忘记释放资源而导致的系统资源泄漏问题。

2.3.3 字段帮助

字段帮助 (Field Assist) 的目的是为最终用户提供界面指导。在设计用户界面时，程序员通常需要为用户提供一些指导信息，例如哪些字段是必填的，字段可以接受的数值类型及取值范围等的说明。字段帮助功能可以简化这一工作。

字段帮助提供两种功能，一种是字段装饰 (Decoration Field)，另一种是内容建议 (Content Proposal) 如图2-11所示。

字段装饰可以让开发者在控件的四角添加图片或颜色作为装饰，以提示用户字段的当前状态，例如用户输入错误时，可以显示一个错误图标；必填字段没有填写时，可以显示一个警告图标等。与查看器相同，开发者随时可以通过字段装饰访问底层的控件并直接操作它。

内容建议则允许开发者在控件附近弹出一个下拉框，并在其中提供一些内容选项供用户选择。这个提示框可以由焦点触发，也可以设置成由某个热键触发。

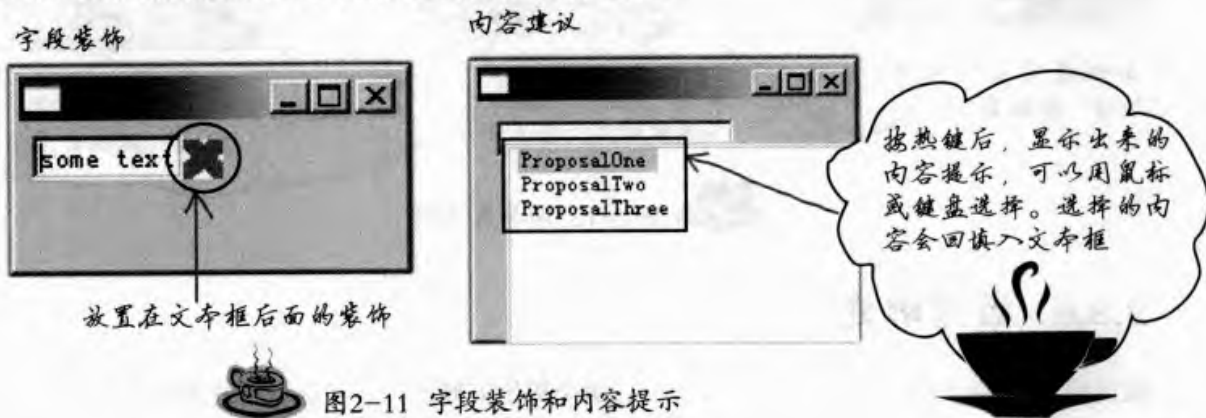


图2-11 字段装饰和内容提示

2.3.4 操作和贡献

操作和贡献 (Action & Contribution) 是用来定制菜单和工具栏的一套机制。这套机制将菜单项/工具栏按钮和它们所触发的事件分离开来，使得对用户操作响应的控制更灵活。

Action对象封装了一个操作命令。当Action对象被添加到工具栏或菜单上面时，用户单击对应的工具栏按钮和菜单项就可以触发这个Action对象所代表的命令。为了可以显示在工具栏或菜单上，Action对象中还包含有图标、工具提示 (tooltip) 等用于显示的信息。

贡献由贡献项目 (Contribution Item) 和贡献管理器 (Contribution Manager) 两部分组成。操作可以用来在菜单或工具栏上添加项目，而利用操作中包含的图标等信息来显示工具栏按钮或菜单项的工作，是由贡献项目完成的。贡献管理器是包含了多个贡献项目的集合。贡献项目代表了菜单或工具栏中的一项，贡献管理器则代表了整个菜单或工具栏。JFace的贡献管理器可以用来控制菜单、工具栏和状态栏。贡献管理器会通过其中包含的贡献项目的信息自动调整这些控件。

关于如何使用操作、贡献管理工具栏和菜单等控件的内容，请见本书的第8章。

2.3.5 对话框、向导页和偏好设置

在SWT的对话框控件基础上，JFace实现了许多定制的对话框。如带有图标对话框、可以在系统托盘中显示的对话框、供用户输入内容的对话框等。

向导页（WizardPage）是用来引导用户一步一步完成某个操作的界面；而偏好设置（Preference）可以用来设计专业的用户选项卡。JFace的框架帮助开发者完成了页面切换的工作，使开发者只需要编写每一页的内容并将它们添加到框架中就可以实现向导页或偏好设置的功能。这两部分内容在Eclipse IDE中广为使用。在Eclipse中创建一个Java工程时，使用的就是向导页，如图2-12所示。

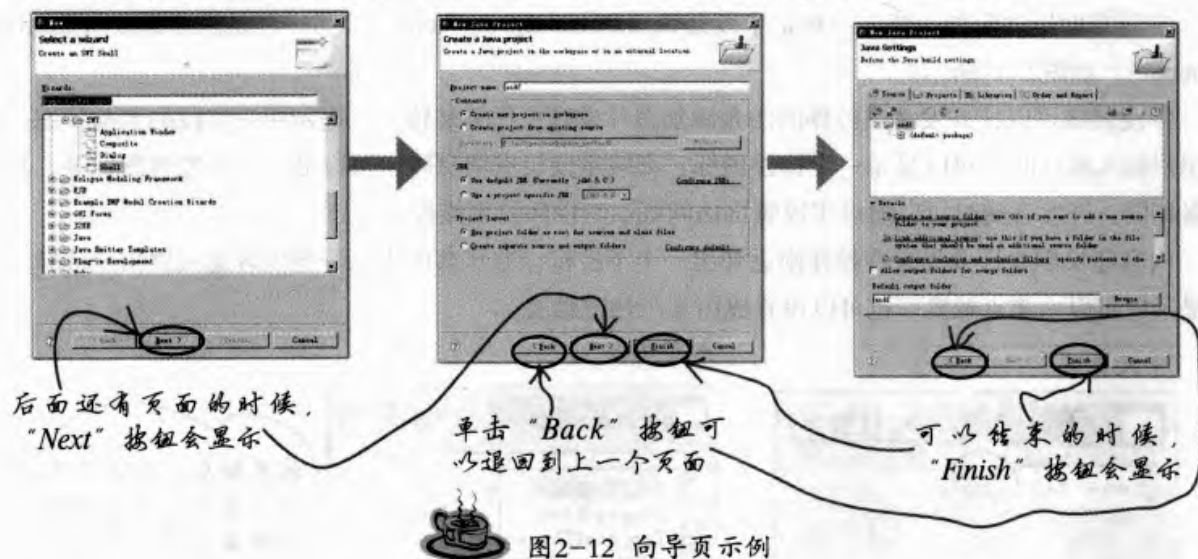


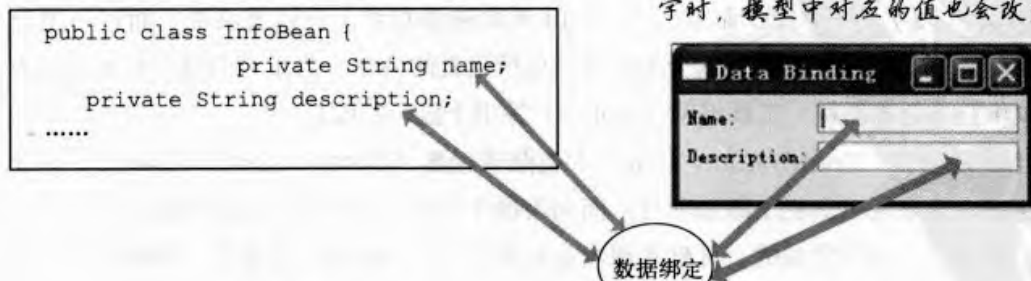
图2-12 向导页示例

2.3.6 数据绑定

数据绑定（Data-binding）是在JFace 3.2版中提出的概念，其实这只不过是把查看器的概念扩展到了整个界面。查看器是为了方便使用复杂控件而将其数据模型抽象出来，数据绑定技术就是将整个图形界面（一个窗口、一个对话框等）看做了一个复杂控件，并将它关联到了一个数据模型上。数据绑定功能负责监听界面上所有控件和数据模型的内容，无论哪一方发生变化时都即时通知另一方。数据绑定示例如图2-13所示。

当模型变化时，文本框的显示内容会随之改变

当用户在文本框中输入或修改文字时，模型中对应的值也会改变



将页面上的两个文本框绑定到了数据模型类中的两个String属性



图2-13 数据绑定示例

使用数据绑定技术，开发者在编写完图形界面，并创建绑定后，数据绑定就会负责根据变化自动同步数据模型和界面视图的内容。当开发者需要取得用户输入时，可以直接访问数据模型的对应部分并获取所关心的信息，而不需要每次都编写访问界面控件的代码；当程序要将数据展现给用户时，也只需要更新数据模型就可以了。这减轻了开发者重复编写界面访问代码的工作，从而提高了开发的效率。

2.4 SWT与Swing

SWT与Sun的AWT/Swing源自两种完全不同的设计思想，因此也走上了截然不同的发展道路。两者各有所长，又各有所短，究竟孰优孰劣，一时难有定论。

Swing是Sun公司所设计的一套图形工具箱，它的目标同样是打造一套跨平台的图形界面开发工具。在SWT之前，Java在桌面程序领域一直举步维艰。这很大程度上是由于Sun公司对图形界面程序的设计理念所造成的。众所周知，每种图形平台都有自己与众不同的显示风格，Windows是蓝底白字的标题栏和浅灰色的背景；Linux是铁灰色的背景；而MacOS X则是彩色的水晶图标和白色的背景。然而Sun却希望将这些风格调和在一起。从最早的AWT到后来的Swing，Sun一直坚持着“界面跨平台”的理念，图形界面程序无论在哪一个平台下面运行，风格都要保持不变。用户可以通过在Swing提供的显示风格中选择他们喜欢的类型而设置程序的风格。Sun希望，这种做法能够使得用户无论在哪一个平台上工作，都能选择使用他们熟悉的同一种界面风格，从而达到无缝地在平台间切换的效果。在Swing中，这种风格被称为LookAndFeel (L&F，界面风格与使用体验)。如图2-14所示的是Swing的几种LookAndFeel在Windows平台上的显示效果。



图2-14 Swing的LookAndFeel

这种想法的初衷的确是从用户角度出发，却给图形工具箱的开发带来了极大的压力。在每个平台上界面显示风格都要一样，就意味着Swing不能直接使用操作系统提供的图形显示功能而只能“模拟”它们。换句话说，Swing必须自己“画”出和目标操作平台的风格相似的界面。

Swing采用了使模型和显示分离的方法。控件本身并不包含任何和绘图相关的代码，而是将绘图代码分离到另外的类——UI类中。UI类的结构和Swing控件的类型结构基本上是一一对应的。比如和JButton (Swing中的按钮控件) 类型相对的UI类是ButtonUI，而和JTextbox (文本框控件) 对应的UI类则是TextUI。Swing将这些UI类声明为抽象类，并要求每种LookAndFeel都实现自己的UI类

型，在其中添加具体负责绘制控件的图形代码。图2-15以按钮控件的绘制为例演示了Swing的这一机制。

控件的绘图过程是由Swing中被称为UI管理器的模块控制的。当程序接到绘图请求时，会将这一请求发给UI管理器。管理器在确定当前使用的LookAndFeel类型后，会根据需要绘图的控件类型获得对应的控件UI类型，然后使用这个UI对控件进行重绘。如果程序员需要改变显示风格，只需要在UI管理器上调用setLookAndFeel方法重设显示风格，这样下一次绘图时，UI管理器就会使用新的L&F了。

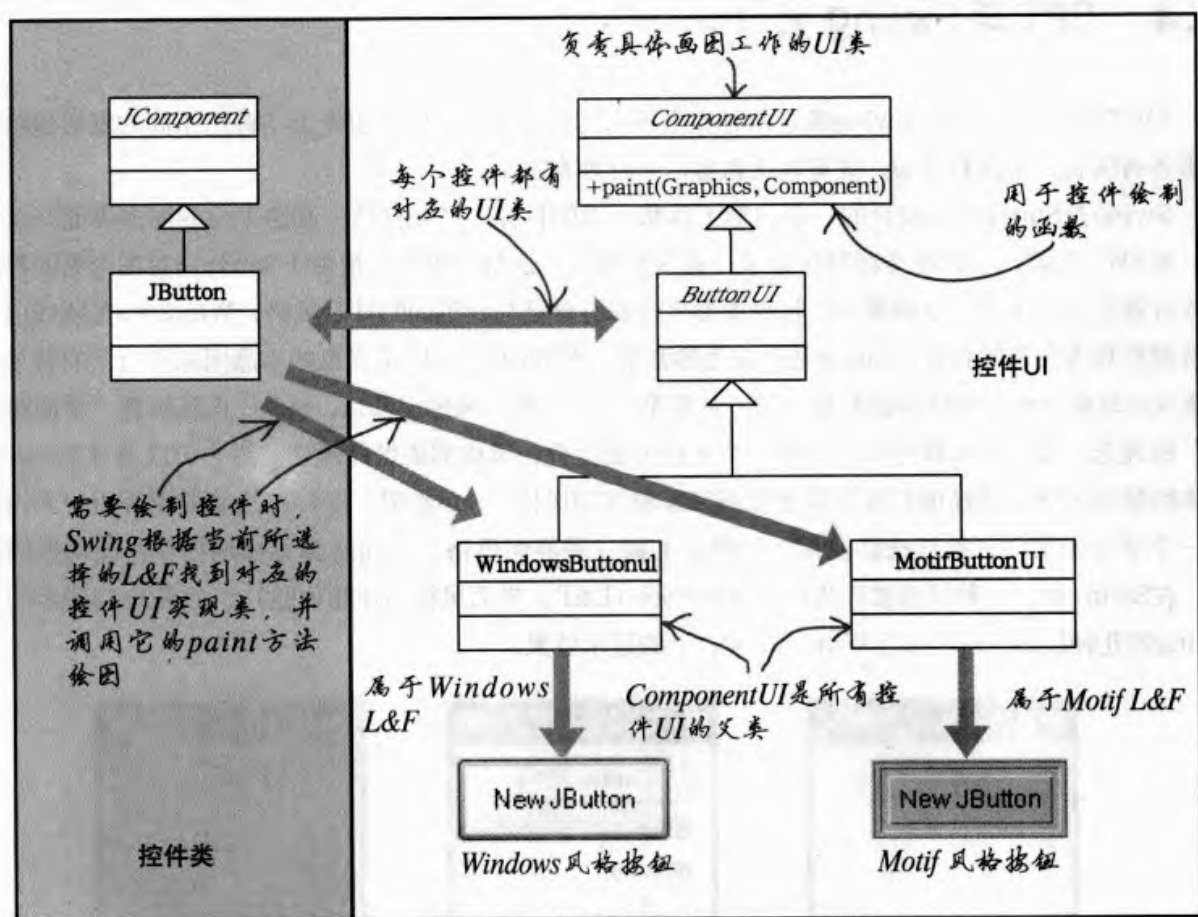


图2-15 Swing的UI机制

Swing设计的构思的确精巧，而且有着其不可取代的优势，比如说，使用Swing，开发者可以自定义控件的绘制过程，甚至画出他们自己设计的控件；对控件的操作全部使用MVC的结构使得编程工作简单高效等。然而它还是存在着一些不足，例如，和本地程序使用的控件相比，画出来的控件在图形效果，如3D效果等方面处于劣势；而且在性能上也无法和使用操作系统API创建的本地控件相比。

谈论Swing和SWT孰优孰劣本身没有太大意义。因为两者的设计目的从一开始就不相同，也有着各自不同的目标用户。开发人员必须意识到，究竟是选择SWT还是Swing，取决于程序的设计目的和最终用户的需要。

2.5 编写并发布SWT程序

2.5.1 第一个SWT程序

为了开发SWT程序，首先要获取一个SWT的开发包。在www.eclipse.org/swt/可以找到对应不同平台的最新版SWT开发包。目前的稳定版本为3.2，以下的说明均以SWT Win32平台3.2.2版为例。

将下载的zip文件解压到任意目录。在以下的说明中，这个目录均以SWT_ROOT代表。关键的文件有如表2-2所示几个。

表2-2 SWT开发包中的关键文件

名称	描述
swt.jar	包含着SWT Java class
src.zip	SWT的源代码，在调试程序时会很有用
数个DLL文件	动态链接库文件

在Eclipse中创建一个新项目，命名为“HelloSWT”。将SWT_ROOT/swt.jar加入BuildPath。为了方便调试代码，还要指定源文件的路径，右键单击swt.jar，选择“属性”，在属性设置的“Java源代码连接”中将src.zip设成swt.jar的代码来源，如图2-16所示。

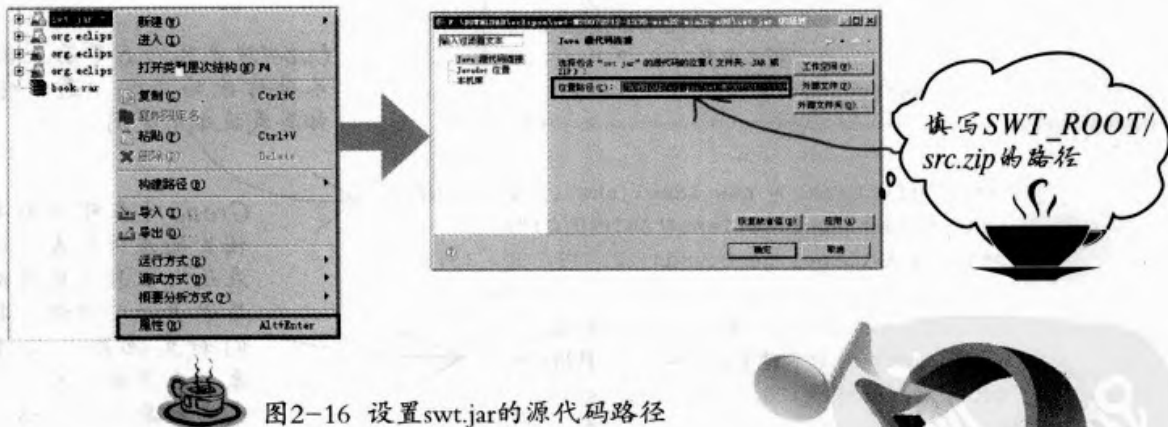


图2-16 设置swt.jar的源代码路径

第一个SWT程序代码如下所示。

```
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class HelloSWT extends Shell {
```



```
private static Text text;
private static Button swtButton;
private static Button swingButton;
private static Button awtButton;
private static Group group;
private static Button button;
private static Label benefitOfSwtLabel;
private static List list;
```

这些都是在下面将要用到的控件

```
public static void main(String[] args) {
```

```
    Display display = Display.getDefault();
    final Shell shell = new Shell(display);
    shell.setText("Hello SWT");
    shell.setSize(260, 283);
```

创建一个`Display`对象，并使用这个`Display`创建一个窗口（`Shell`），并设置它的标题和窗口初始尺寸。因为当前线程创建了`Display`对象，所以它是界面线程

```
    shell.open();
```

```
    text = new Text(shell, SWT.BORDER);
    text.setText("SWT是Eclipse平台使用的图形工具箱");
    text.setBounds(10, 8, 230, 35);
```

创建一个`Text`控件并设置其中文字的内容。然后设置了控件在窗口中的位置，左上顶点为(10,8)，宽度为230，高度为35

```
    list = new List(shell, SWT.BORDER);
    list.setItems(new String[] {
        "使用操作系统本地控件",
        "提供一套平台无关的API",
        "GUI程序的运行速度快",
        "更多更多....." });
    list.setBounds(10, 68, 232, 82);
```

`List`控件可以用来显示一系列的内容。这里创建了一个有4条内容的`List`控件

```
    benefitOfSwtLabel = new Label(shell, SWT.NONE);
    benefitOfSwtLabel.setText("SWT的优点:");
    benefitOfSwtLabel.setBounds(10, 49, 90, 15);
```

`Label`控件可以在界面上显示不能修改的文字，通常作为标签或提示来使用

```
    group = new Group(shell, SWT.NONE);
    group.setText("你使用过哪些图形工具箱?");
    group.setBounds(10, 159, 230, 47);
```

`Group`对象可以用来把相关的控件包成一组，在这一组控件的外面会显示出一个边框，将它们和其他控件隔离开来。在边框上可以加入文字标题以说明这一组控件的作用

```
    awtButton = new Button(group, SWT.CHECK);
    awtButton.setText("AWT");
    awtButton.setBounds(10, 20, 54, 18);

    swingButton = new Button(group, SWT.CHECK);
    swingButton.setText("Swing");
    swingButton.setBounds(70, 22, 60, 15);
```

`Button`类型包含普通按钮，单选按钮，复选框等很多形式，这里使用的是复选框

```

swtButton = new Button(group, SWT.CHECK);
swtButton.setBounds(136, 22, 62, 15);
swtButton.setText("SWT");

button = new Button(shell, SWT.NONE);
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(final SelectionEvent e) {

        MessageBox msgBox =
            new MessageBox(shell, SWT.ICON_INFORMATION);
        msgBox.setMessage("Hello SWT!");

        msgBox.open();

    }
});
button.setText("按一下按钮,向SWT说Hello");
button.setBounds(10, 214, 227, 25);
shell.layout();

while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
        display.sleep();
}
}

```

在这里创建了一个普通按钮。并为其添加了一个选择事件监听器。按钮被按下时，会触发选择事件 (SelectionEvent)

在选择事件监听器中，创建了一个对话框，并且在其中显示一条“HelloSWT”的消息。按下按钮的时候，这个对话框就会显示出来

程序运行结果如图2-17所示。



运行SWT程序的方式和普通的Java程序有所不同。右键单击需要以SWT模式运行的Java文件，在“运行方式”菜单中，不要选择通常所选的“Java应用程序”，而代之以“SWT 应用程序”。这样就可以方便地运行SWT程序，否则运行时就会出现如图2-18所示的异常。

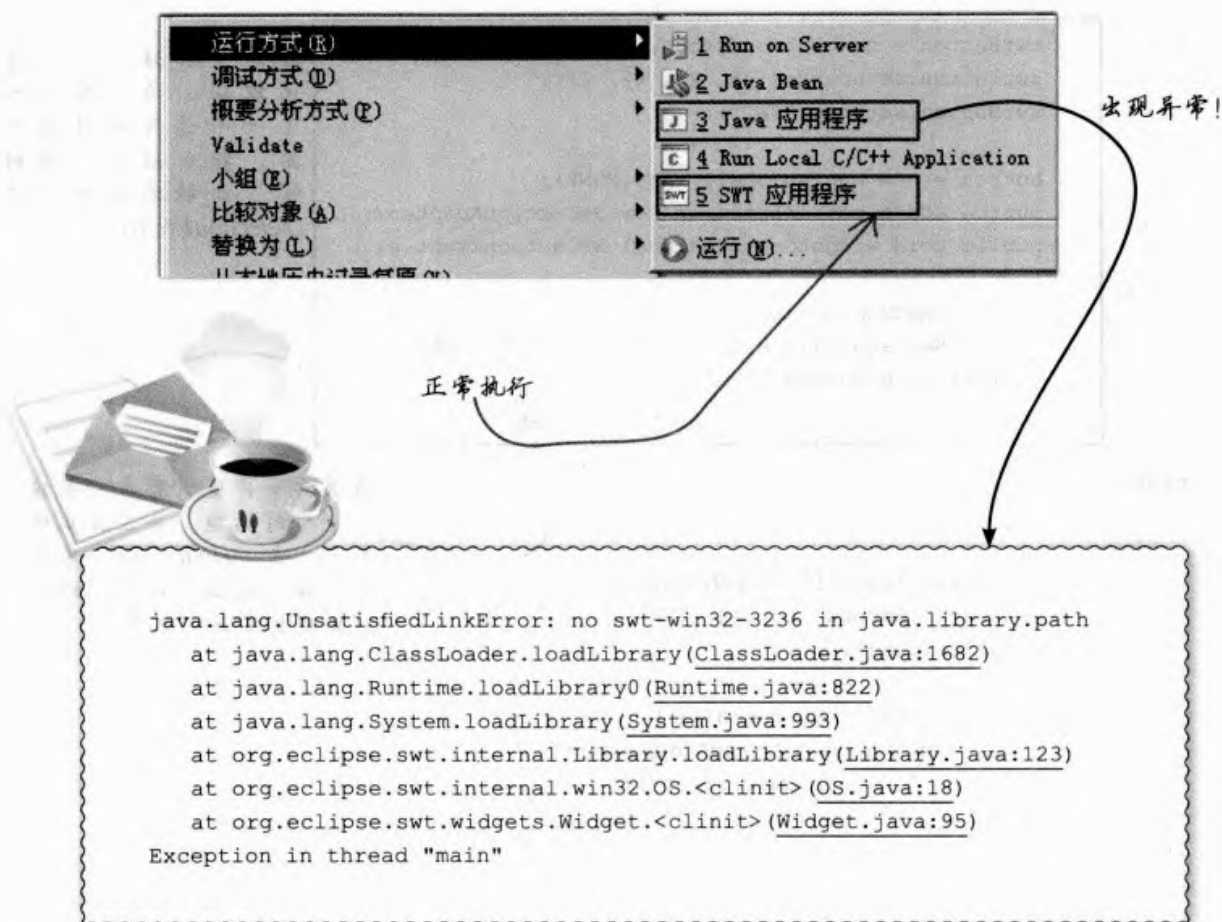


图2-18 在Eclipse IDE中运行SWT程序

出现该异常的原因是SWT使用了JNI技术和底层操作系统进行交互，因此在运行时，需要在Java系统变量java.library.path中指定几个动态链接库（在Windows系统下是DLL文件，在Linux系统下则是lib.so文件）的路径才能正常运行，否则就会报出这样找不到动态链接库文件的异常信息。右键单击swt.jar，选择“属性”→“本机库”，并选择DLL所在的目录，即SWT_ROOT，如图2-19所示，单击“确定”按钮后，再次运行程序，就可以使用“Java运行方式”正常执行了。

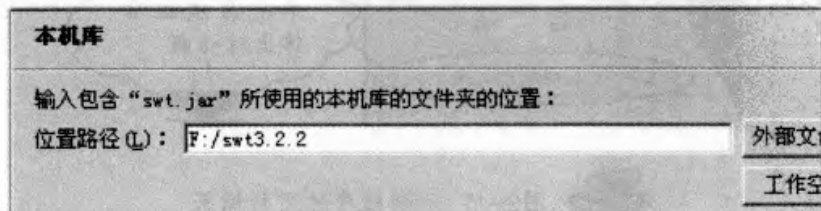


图2-19 为JAR包设置本机库路径

另外一个比较方便的途径就是使用“SWT应用程序”方式运行。在这种方式下，Eclipse会使用平台自带的SWT动态链接库运行程序，免除了手动配置路径的步骤。

2.5.2 SWT程序的打包发布

本节会将前面创建的程序打包发布，使其能够脱离Eclipse单独运行。首先，需要将完成的SWT项目从Eclipse中导出成JAR文件。选择HelloSWT项目并单击右键，选择“导出”，在弹出的“导出”对话框中选择“Java”→“JAR文件”，如图2-20所示。

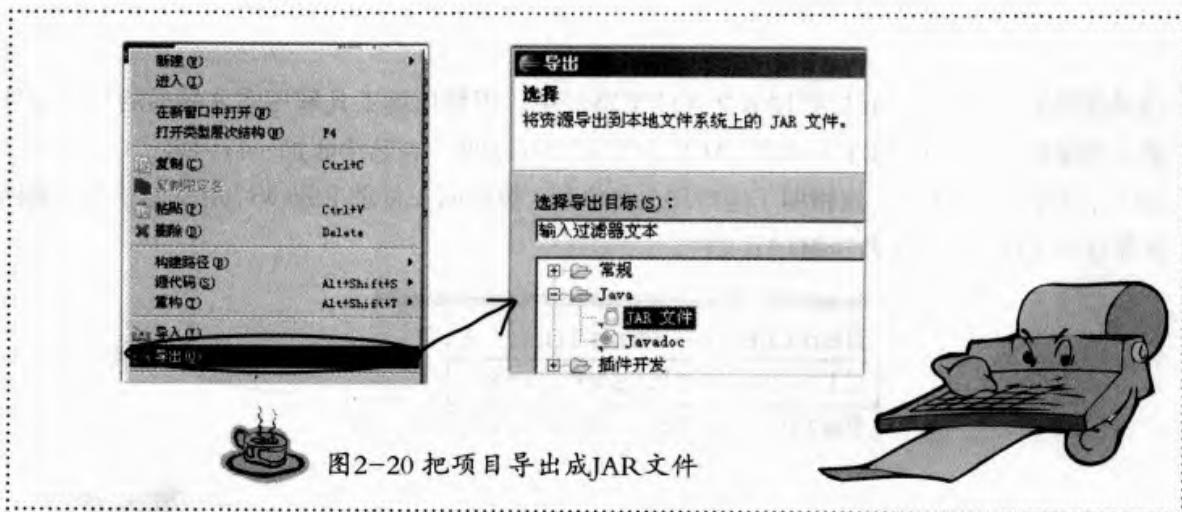


图2-20 把项目导出成JAR文件

在下一步的“JAR导出”对话框中，选择需要导出的项目（可以展开项目后查找并选择是否要导出项目中的某一个文件）并选择所要导出的文件类型。常用的主要有两个选项，导出类文件和导出源文件。如果只希望导出可执行代码，选择第一项“导出生成的类文件和资源”就可以了。如果希望将源代码同时发布出去，就同时选择第三项“导出Java源文件和资源”，如图2-21所示。在“选择导出目标”处输入JAR文件路径和文件名。选择一个目录并为JAR文件命名为“helloSWT.jar”后，一直单击“下一步”按钮来到最后一个页面。在“选择应用程序入口点的类”一项中，选择包含程序执行入口的main函数的类为HelloSWT。只有这样指定了入口点后，才可以直接用java.exe命令加-jar的参数执行这个JAR文件。

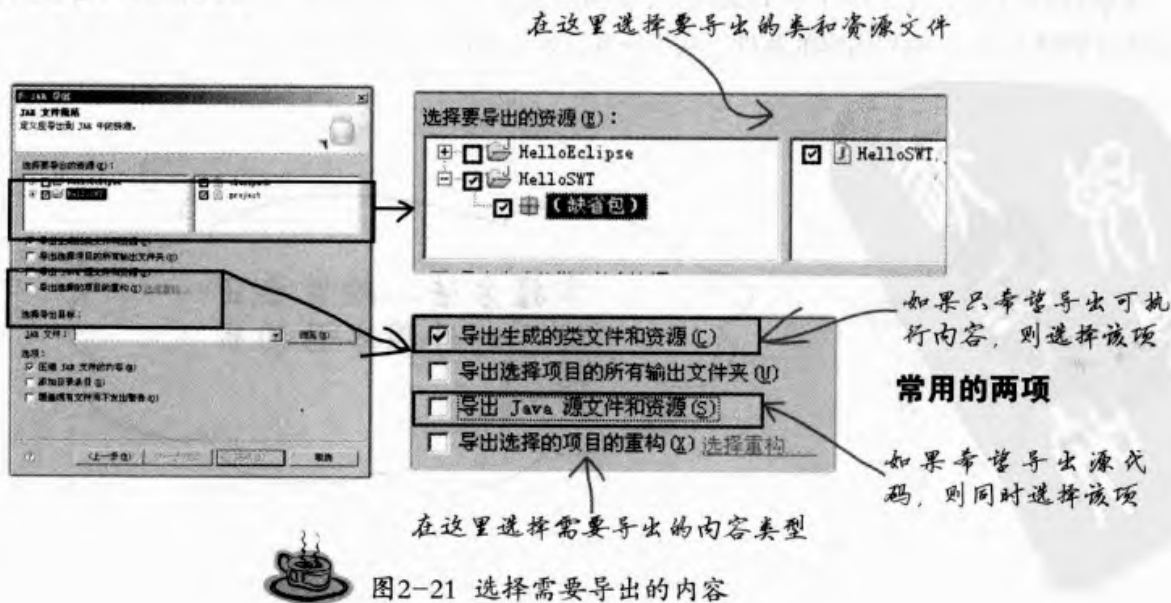
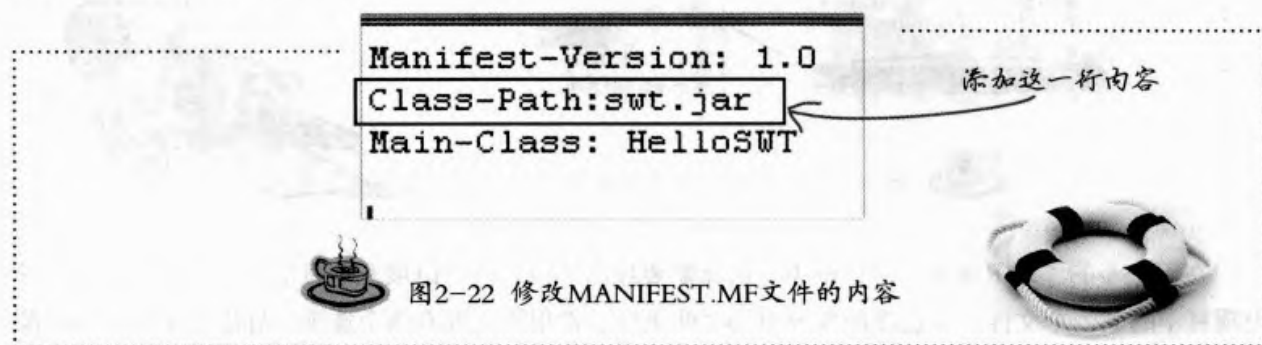


图2-21 选择需要导出的内容

现在可以尝试运行这个JAR文件。在命令行方式下执行命令“java-jar helloSWT.jar”，会得到一个NoClassDefFound的错误，如下所示。

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/eclipse/swt/widgets/Shell
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:620)
    .....
```

这是因为没有将swt.jar加到JAR文件的类路径中。用解压缩工具解压刚才导出的JAR文件，用任意文本编辑工具打开META-INF\MANIFEST.MF文件，在其中添加一行内容“Class-Path: swt.jar”，如图2-22所示。这指明了运行JAR文件时，要将同一目录下的swt.jar文件添加到类路径中。保存这个文件，然后重新压缩JAR文件。



现在在命令行中再次执行“java-jar helloSWT.jar”，就可以正常执行该 SWT程序了。

2.6 本章小结

本章对SWT的架构、SWT API的结构、JFace的内容都做了较为概要的介绍，这些理论知识是后面学习SWT与JFace编程内容的基础。从下一章开始，将进入SWT实战的阶段。



掌握方法，投篮也变成了一件很容易的事情了！

第3章 SWT编程基础

拉开崭新的学习帷幕

上一章对整个SWT/JFace的体系架构作了简要的介绍，了解了这些理论知识后，读者需要通过更多的实践来深入体会它们。本章将开始学习SWT编程的基础内容，包括窗口、控件及图形资源等部分。通过本章的学习，读者可以使用SWT创建一个窗口，认识SWT是如何与操作系统交互的，并且了解SWT所提供的控件、可使用的图形资源及SWT中的资源管理机制等。

本章内容包括：

- ★ Display和Shell。
- ★ 控件。
- ★ 图形资源。
- ★ 高级内容。



进入第03章

3.1 Display和Shell

编写图形界面程序的第一步，是要为界面创建一个窗口。下面的代码创建了一个空白窗口(代码见光盘：\book.ch3.BlankWindow.java)。

```
public static void main(String[] args) {
    1 Display display = Display.getDefault();

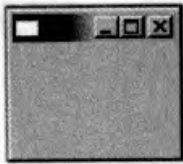
    2 Shell shell = new Shell(display);
      shell.setSize(100, 100);
      shell.open();
      shell.layout();

    3 while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
            display.sleep();
    }
}
```

创建一个窗口对象

设置窗口尺寸

打开窗口并进行布局



所创建的空白窗口

这个程序可以分为三大部分，1：创建Display对象；2：创建并配置Shell；3：进入一个循环。在下面几节中将逐一讲解它们所起的作用。

3.1.1 Display的创建

在示例程序的开始，首先得到了一个Display对象。这是所有SWT程序开始运行时都必须执行的工作。没有Display，SWT程序就无法和操作系统交互，一个SWT程序中至少需要一个Display对象。创建Display对象的线程被称为UI线程。

Display类提供了getDefault()方法。首次调用这个方法会创建一个Display实例，随后再次调用这个方法可以获取已创建的实例。在示例程序Blank Window中，由于main线程调用了Display.getDefault()方法，它就是所得到的Display对象的UI线程。下面的两段代码给出了Display.getDefault()通常的使用方式。

```
.....//UI线程启动
Display createDisplay = Display.getDefault();
.....// 使用Display实例
Display display = Display.getDefault();

.....// UI线程启动
Display newDisplay = new Display();
.....// 使用Display实例
Display defaultDisplay = Display.getDefault();
```

第一次调用getDefault，创建一个Display

第二次调用getDefault，得到上面创建的Display

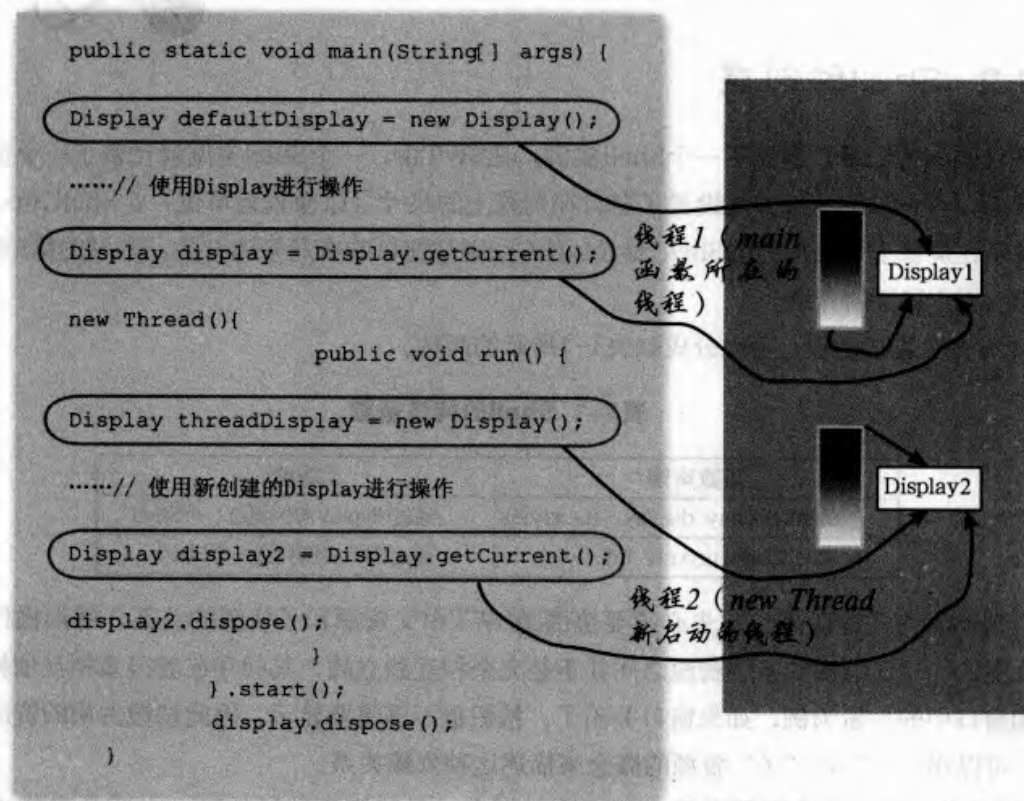
调用构造函数创建一个Display，如果现在没有一个默认的Display，这个Display就会被设置成默认

调用getDefault，会得到上面创建的Display

同一个对象

同一个对象

一个线程中不能同时有两个活动的Display存在，如果试图这样做（比如连续新建两个Display对象），程序在运行时会抛出一个SWT异常；如果某些程序由于特殊的需求，需要创建多个同时活动的Display实例，则必须在不同的线程中创建它们。使用Display.getCurrent()函数可以得到当前线程对应的Display实例，而Display.findDisplay(Thread)则可以找到任意线程，对应的Display实例代码如下所示。



涉及到多个线程时，不应该使用Display.getDefault来得到当前线程所属的Display实例。这个方法在多UI线程的情况下很容易出现错误，下面的代码演示了一种常见的错误用法。



可以看到，在不同的线程中调用Display.getDefault()方法返回的是同一个Display实例。如果不加判断而直接使用它的话，很容易产生“非法线程访问”的SWT异常。在开发多UI线程的程序时，推荐使用如下代码来管理Display实例。

下面的一段代码(代码见光盘: \book.ch3.ShellDependency.java)演示了这种依赖关系。

```
public static void main(String[] args) {
    Display display = Display.getDefault();

    Shell shell1 = new Shell(display);
    Shell shell2 = new Shell(shell1);

    shell1.setText("Parent Shell");
    shell2.setText("Child Shell");

    shell1.setSize(200,200);
    shell2.setSize(200,200);

    shell1.open();
    shell2.open();
    while (!shell1.isDisposed()) {
        if (!display.readAndDispatch())
            display.sleep();
    }

    display.dispose();
}
```

生成两个Shell,
shell2(Child Shell)
是shell1(Parent
Shell)的子资源



代码创建了两个Shell,一个是从属于Display的“Parent Shell”,另外一个是从属于“Parent Shell”的“Child Shell”。运行后,会显示两个窗口。关闭Child Shell的窗口,Parent Shell不受影响;而关掉Parent Shell的时候,Child Shell也被关闭了。

用Shell.open()方法打开一个窗口后,它就会显示出来。使一个窗口消失有两种方法,一种是调用Shell.close(),这种方式会关闭窗口,并释放掉其占用的系统图形资源;另一种则是使用Shell.setVisible(false),这种方式只是暂时隐藏窗口,不会释放所占资源,调用Shell.setVisible(true),就可以让窗口再次显示出来。

3.1.3 Display的事件队列和事件循环

创建了一个窗口并打开它以后,示例就进入事件循环(Event Loop)部分,代码如下所示。

```
while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
        display.sleep();
}
```



在分析事件循环的代码之前,首先需要对Display的事件处理机制有一个整体认识。常见的图形操作系统会为每一个GUI程序分配一个事件队列,用户操作鼠标或键盘时,操作系统负责将这些事件放到对应程序的事件队列中。

但是除了系统事件之外,程序可能还希望在事件循环中处理一些自定义的事件。为此,Display

额外维护了一个应用程序级别的事件队列，自定义的事件可以被添加到这个队列中。在Display的事件循环中，同时处理着系统队列和这个自定义队列中的事件，如图3-2所示。在后面的内容中将讲到如何使用这个自定义队列。

现在开始分析事件循环的代码。第一行代码很容易理解，如果Shell没有被释放的话，事件循环就要一直继续下去。

下面代码是readAndDispatch()方法的内容，这里调用了很多org.eclipse.swt.internal.win32.OS类上面的方法。如果读者有Win32的编程基础，可能会觉得这些方法的名称有些眼熟。OS是SWT用来包装操作系统API的类型，它的函数都是JNI的接口，而且与操作系统的API一一对应。



图3-2 Display中的事件队列

```
public boolean readAndDispatch () {

    checkDevice ();
    drawMenuBar ();
    runPopups ();

    if (OS.PeekMessage (msg, 0, 0, 0, OS.PM_REMOVE)) {

        if (!filterMessage (msg)) {
            OS.TranslateMessage (msg);
            OS.DispatchMessage (msg);
        }

        runDeferredEvents ();
        return true;
    }

    return runMessages && runAsyncMessages (false);
}
```

从系统队列中取出一条消息

关于Win32 API的知识，读者可以查阅MSDN。在MSDN中，关于Win32程序的消息循环有如下的一段示例代码（C代码）。

```
// Start the message loop
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```



两段代码主要的程序架构几乎是一模一样，首先从事件队列中将事件取出来，经过必要的翻译

后(TranslateMessage)发送到窗口处理(DispatchMessage)。小小的差别在于读取事件队列时,MSDN的例子中使用了 GetMessage,而SWT的消息循环中使用了 PeekMessage,关于两者的区别,MSDN中有如下面注意中所述的说明。

※ 注意: ※



PeekMessage与GetMessage的作用类似,两者都会检查事件队列中是否有事件存在并将它们拷贝出来。两者的主要区别在于GetMessage在读不到事件时会一直等待下去并阻塞线程直到事件队列中有事件为止;而PeekMessage在没有事件时会立刻返回。

在调用操作系统API转发消息之前,SWT还使用filterMessage方法对消息事件做了过滤。在通常情况下,发送到窗口中控件的事件都会交由窗口的消息处理函数统一处理。如果事件是发送到某一个特定控件(如按钮、菜单等)的,这个方法允许控件“吃掉”这个事件而不发送到主窗口。如果在菜单和它的窗口中为同一个快捷键设置了不同的功能,打开菜单时,快捷键消息就会被菜单捕捉下来而不发送到窗口。

SWT的事件循环中,方法runDeferredEvents()和runAsyncMessages()负责处理Display的自定义事件队列中的事件。

现在可以理出Display.readAndDispatch的流程,首先从系统事件队列中读取消息,如果在程序的事件队列中读到事件,就将它发送到窗口去处理;如果在线程交互的事件队列中有需要执行的事件,就去执行它。图3-3演示了这一流程。

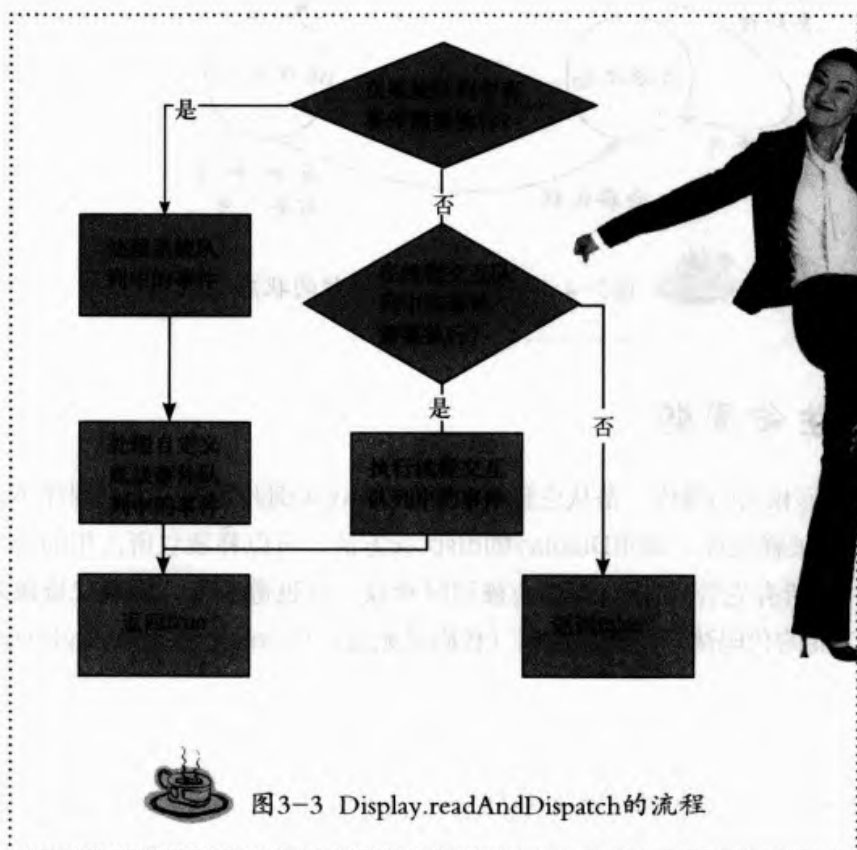


图3-3 Display.readAndDispatch的流程



如果readAndDispatch返回false，事件循环就会调用Display的sleep方法。sleep方法内容如下所示。

```
public boolean sleep () {
    checkDevice ();
    if (runMessages && getMessageCount () != 0) return true;

    if (OS.IsWinCE)
        OS.MsgWaitForMultipleObjectsEx (0, 0, OS.INFINITE,
        OS.QS_ALLINPUT, OS.MWMO_INPUTAVAILABLE);
    return true;

    return OS.WaitMessage ();
}
```

该段代码很明显
是针对WinCE操
作系统

略过中间一段针对WinCE的代码，可以注意到sleep方法调用了OS.WaitMessage。WaitMessage会使当前线程（UI线程）休眠，这样就可以将处理器让给别的线程使用。当事件队列中有新的事件传来时，UI线程会被唤醒并恢复事件循环过程。如图3-4所示的状态图演示了整个事件循环中UI线程的活动/休眠状态变化。

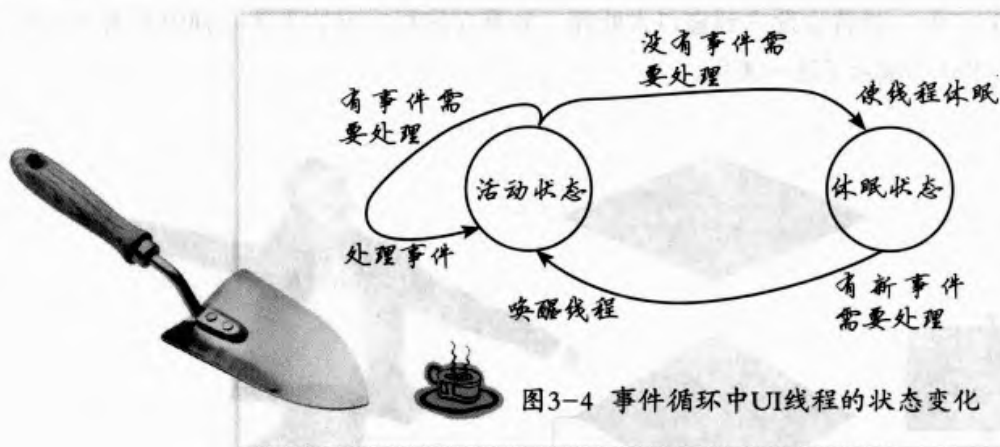


图3-4 事件循环中UI线程的状态变化

3.1.4 Display的生命周期

一个SWT程序真正开始图形相关的操作，是从它创建一个Display实例开始的，而当程序不再需要这个Display实例时，它就会被释放掉。调用Display的dispose方法，可以释放它所占用的资源。当一个Display实例被释放时，所有它管理的Shell都会被同时释放。这也遵循着“释放父资源时子资源同时被释放”的法则。下面的代码演示了这一法则（代码见光盘：\book.ch3.DisplayDispose.java）。

```
public static void main(String[] args) {
    Display display = Display.getDefault();
}
```

```
Shell shell1 = new Shell(display);
shell1.setSize(200, 100);
Button button = new Button(shell1, SWT.NONE);
button.setText("Dispose Display");
button.setBounds(10, 10, 120, 20);
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        Display.getDefault().dispose();
    }
});
shell1.open();
```

在窗口中添加一个按钮，单击这个按钮时，就将UI线程的Display实例释放掉



```
Shell shell2 = new Shell(display);
shell2.setSize(200, 100);
shell2.open();
```

```
while (!(shell1.isDisposed() && shell2.isDisposed())) {
    if (!display.readAndDispatch())
        display.sleep();
}
display.dispose();
}
```

执行这段代码，可以发现，使用关闭按钮关闭（释放）任意一个窗口都不会影响另外一个。而单击“Dispose Display”的按钮后，两个窗口都被释放了。

在本章开始的示例代码中创建了一个Display对象，但是并没有释放它，这事实上是一个特例。在示例代码中，JVM只有一个线程（main函数的线程），当它退出时，JVM的进程也就退出了。而当操作系统中一个进程退出的时候，它所占用的所有资源都会被回收，所以即使没有调用Display的dispose方法，也不会导致系统资源泄漏。然而，这样编写代码并不是一个好的习惯，在通常的编程实践中，还是推荐写成如下的形式。

```
Display display = Display.getDefault(); //使用Display
display.dispose();
```

创建Display对象，开始使用

使用完后释放Display对象



3.1.5 监视器、边界和客户区域

使用Display.getMonitors()方法可以取得与这个Display相连的所有监视器信息，而Display.getPrimaryMonitor()则可以得到主监视器对象。监视器对象主要包含边界和客户区域两个部分。边界代表了这监视器的屏幕大小，而客户区域则代表可以用来显示窗口的屏幕部分大小。一般来说，由于图形操作系统桌面上的任务栏占掉了一部分空间，客户区域的尺寸要小于边界的尺寸，如图3-5所示。下面一段代码（代码见光盘：\book.ch3. MonitorInfo.java）演示了如何取得主监视器的边界和用户区域大小。

```
public static void main(String[] args) {
    Display disp = Display.getDefault();
```

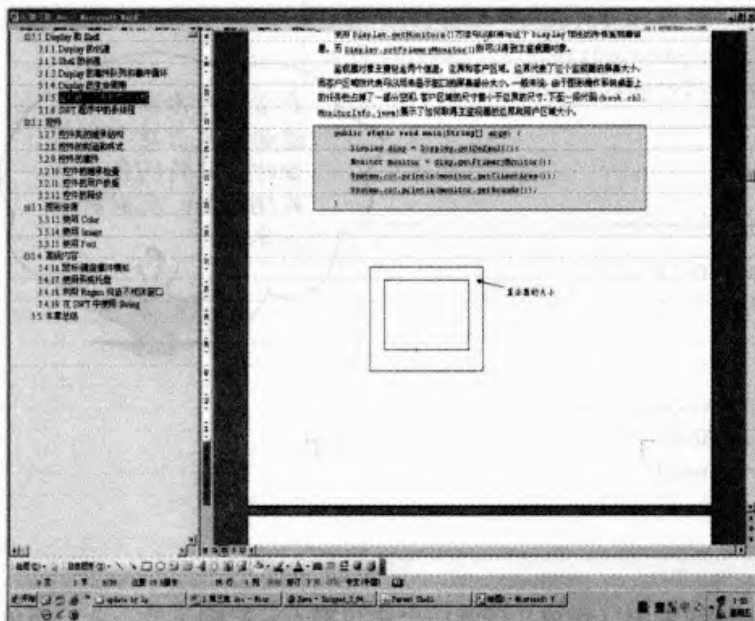


```

Monitor monitor = disp.getPrimaryMonitor();
System.out.println(monitor.getClientArea());
System.out.println(monitor.getBounds());
}

```

调用`getPrimaryMonitor`取得主监视器的信息，如果系统中安装了多个监视器可以用`getMonitors`方法得到它们



监视器的大小代表了整个桌面的大小。如监视器分辨率调为 1024×768 ，则监视器宽为1024像素，高为768像素

客户区域（图中不包含任何程序的窗口区域）指监视器内真正可以用于显示窗口的区域，由于任何程序等系统级组件的存在，客户区域通常略小于监视器尺寸



图3-5 监视器、边界和客户区域示例图

代码执行结果如下：

```

Rectangle { 0, 0, 1280, 964}
Rectangle { 0, 0, 1280, 1024}

```

这说明监视器的当前尺寸是 1280×1024 像素，而可以显示窗口的客户区域为 1280×964 像素（因为Windows XP的任务栏占掉了一部分高度）。使用监视器实例，可以得到当前用户屏幕的大小信息，程序可以根据这些信息安排窗口的尺寸及位置。

3.1.6 SWT程序中的多线程

SWT采用单线程模型管理绘图操作，只有UI线程才能进行控件重绘和处理事件循环等直接访问Display的操作，非UI线程试图直接操作Display则会导致一个SWT异常。

简单的数据计算任务可以直接放在UI线程中执行，但在其中执行比较耗时的操作，却不是一个好的选择。下面的代码对这种情况做了一个演示（代码见光盘：`\book.ch3.UIThread_1.java`）。

```

public static void main(String args[]) {
    try {

```

按下按钮时，让UI线程休眠10秒以模拟耗时的计算，然后改变button的标签以显示完成

```
final Display display = Display.getDefault();
Shell shell = new Shell(display, SWT.SHELL_TRIM);
shell.setText("Multi Thread");
shell.setSize(204, 92);
final Button button = new Button(shell, SWT.NONE);

button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(final SelectionEvent e) {
        try {
            Thread.sleep(10000);
            button.setText("Execution done");
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
    }
});

button.setText("button");
button.setBounds(20, 15, 155, 25);
shell.open();
shell.layout();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
        display.sleep();
} catch (Exception e) {
    e.printStackTrace();
}
```

执行这段程序，单击按钮以后，程序界面停止响应，直到10秒后才恢复正常。如果程序执行一些稍为耗时的计算就会停止响应，这显然是最终用户无法接受的，这就出现了SWT的单线程模型与复杂的程序逻辑的矛盾。为了解决这个矛盾，必须为非UI线程（后台线程）提供一个途径，使它能将需要执行的操作通知UI线程。这个通知动作被称为后台线程与UI线程的同步，如图3-6所示。

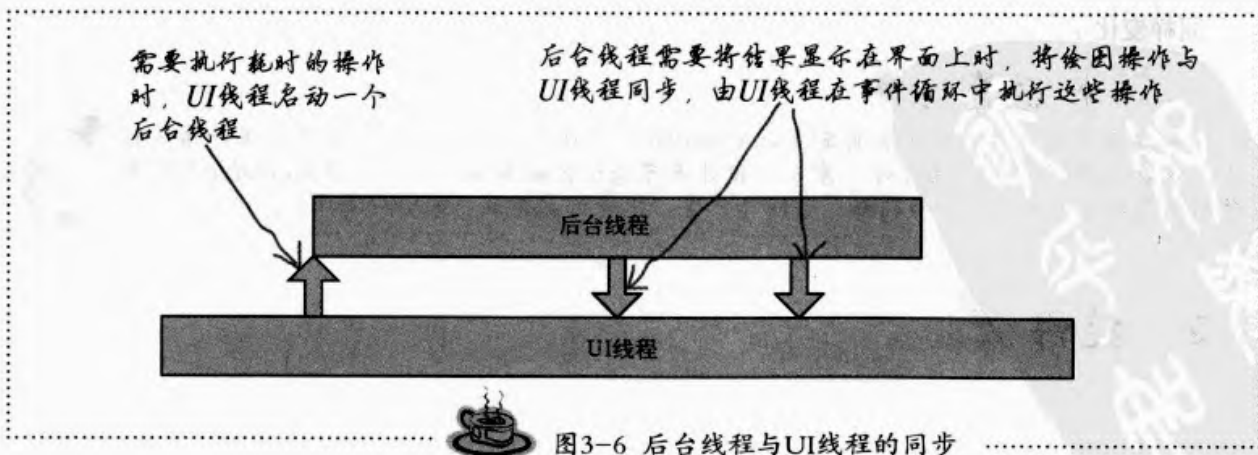


图3-6 后台线程与UI线程的同步

Display维护了一个自定义的事件队列，这个队列就是用来供后台线程和UI线程同步的。后台线程用Runnable对象将绘图操作包装起来，然后将对象插入事件队列中，这样Display执行消息循环时


就会执行这些操作了。Display提供了如下两个方法向这个队列中插入事件。

Display.syncExec(Runnable runnable)	同步调用。调用这个方法会通知UI线程在下次事件循环时执行runnable参数的run方法。调用这个方法的线程将被阻塞到runnable执行完成为止。如果参数是null，调用这个函数会唤醒休眠中的UI线程
Display.asyncExec(Runnable runnable)	异步调用。调用这个方法同样通知UI线程在下次事件循环时执行runnable参数的run方法。调用这个方法的线程不会被阻塞，而且在runnable执行完成后不会得到通知。如果参数是null，调用这个函数会唤醒休眠中的UI线程

现在把本节前面代码中按钮的事件监听器改造一下(代码见光盘：\book.ch3.UIThread_2.java)。

```
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(final SelectionEvent e) {
        Thread thread = new Thread() {
            public void run() {
                try {
                    Thread.sleep(10000);
                } catch (InterruptedException e1) {
                    e1.printStackTrace();
                }
                display.syncExec(new Runnable() {
                    public void run() {
                        button.setText("Execution Done");
                    }
                });
            }
        };
        thread.start();
    }
});
```

创建一个后台线程，休眠10秒后调用Display.syncExec改变button的标签



执行这段代码可以发现，单击按钮后，界面不再出现停止响应的状况，而且10秒钟后按钮的文字同样变化了。

※ 注意 : ※

创建后台线程时，通常会将需要在Runnable的run()方法中访问的图形界面组件传递过去。但在代码被UI线程执行时，涉及的组件有可能已经被释放了。因此在Runnable中需要对所用到的组件的状态加以判断，以防止产生“组件已被释放”的SWT异常。

3.2 控件

3.2.1 控件类的继承结构

所有控件类都从org.eclipse.swt.widget.Control继承而来，如图3-7所示为一些主要控件的继承

结构。

最上层的Widget类直接继承自Object，是SWT中所有和窗口相关的部件的父类，它为所有窗口组件提供了创建对象（申请资源）、释放资源和事件监听的功能。

Widget有两个比较重要的子类，Control类是所有控件的父类；而Item类则是控件的辅助部分，它代表了复杂控件中的某一“项”，如表格控件中的一行就是由TableItem来表示的。

Button、Label等简单控件都直接继承自Control控件；Scrollable代表拥有滚动条的控件，它的子类有Text和Composite等；Composite被称为容器控件，这类控件主要的用途不是与用户交互，而是将容纳于其中的其他控件在界面上分组摆放，它的子类有Table、Tree等可以用来显示复杂数据结构的控件。

Canvas是允许程序员使用画点、线等基本图形操作在其中直接绘图的控件。Shell间接继承自Canvas类。

整个SWT的控件家族极为庞大。在下面的章节中，将对常用控件的使用一一加以介绍。

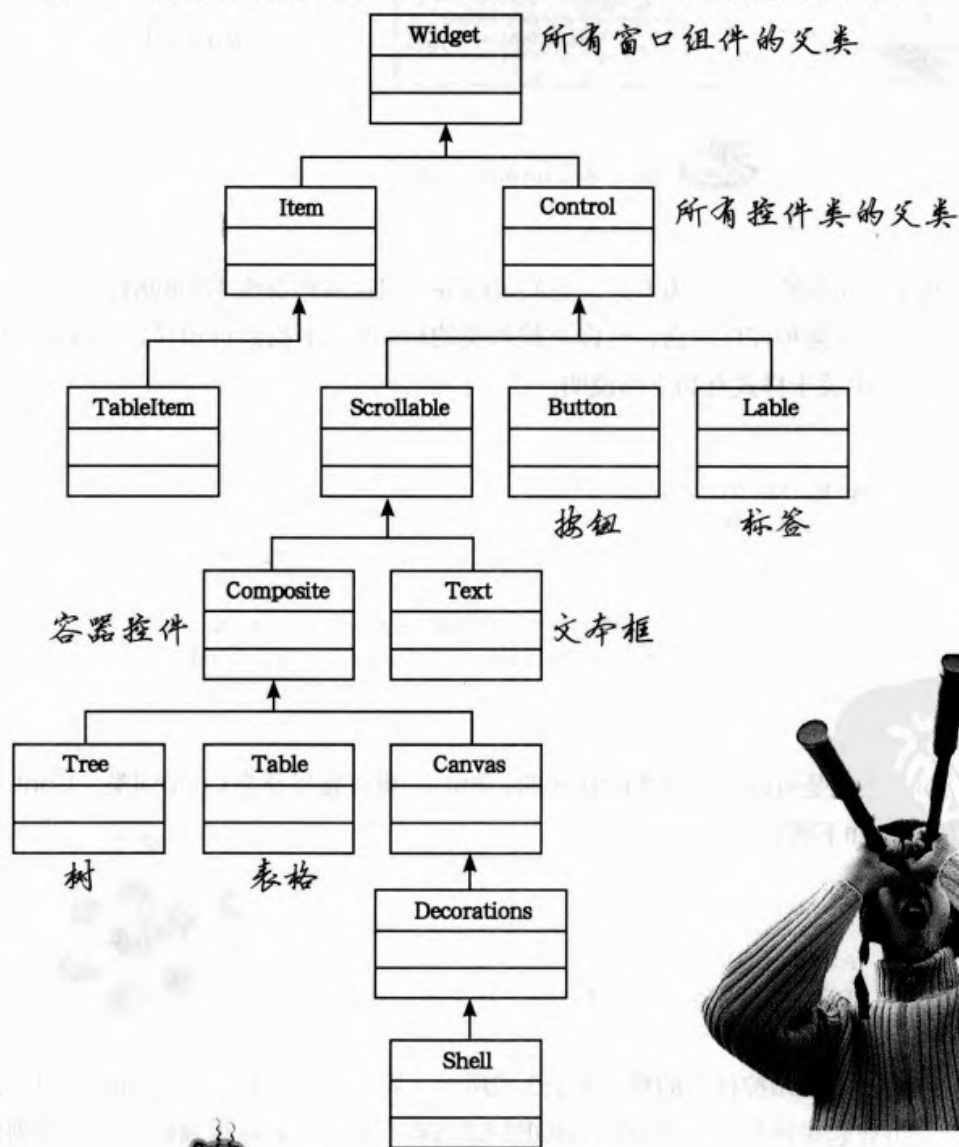


图3-7 SWT 中的控件继承结构



3.2.2 控件的构造和样式

可以用Control(Control parent, int style)来构造一个控件实例,其中第一个参数指明了控件的父资源,第二个参数是控件使用的样式。下面的代码在shell代表的窗口中使用BORDER和PUSH样式创建了一个Button控件。

```
Button button = new Button(shell, SWT.BORDER | SWT.PUSH);
```

构造控件时,必须指定一个父资源,Shell或其他Composite类的控件都可以作为一个控件的

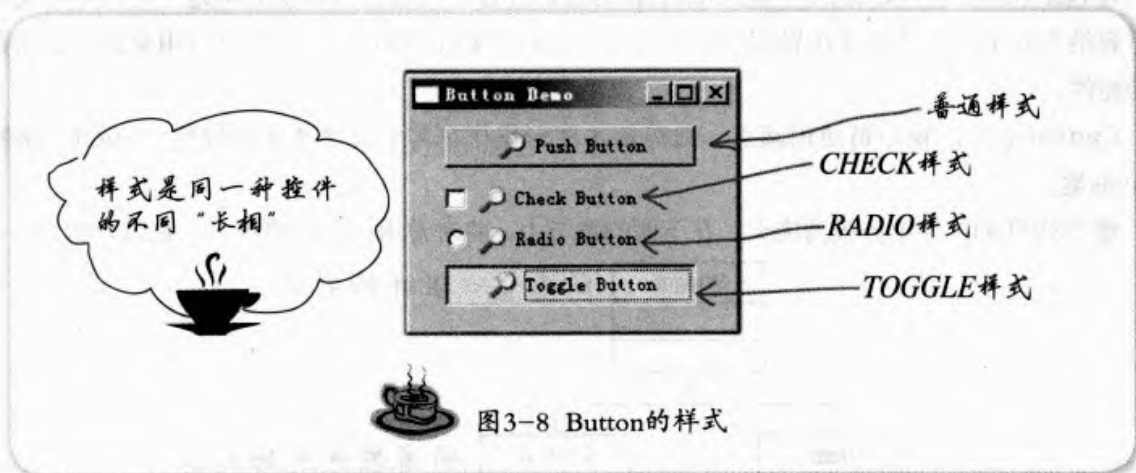


图3-8 Button的样式

样式控制着控件很多方面的显示和行为方式,它们可以按照用途被归类成不同的组,同属一组的样式应用时只能选择一个。对某种控件而言,可以在控件类的JavaDoc中找到可用样式的具体说明,以Button为例,其JavaDoc中关于样式有如下的说明。

Styles:

ARROW, CHECK, PUSH, RADIO, TOGGLE, FLAT
UP, DOWN, LEFT, RIGHT, CENTER

Note: Only one of the styles ARROW, CHECK, PUSH, RADIO, and TOGGLE may be specified.

Note: Only one of the styles LEFT, RIGHT, and CENTER may be specified.

Note: Only one of the styles UP, DOWN, LEFT, and RIGHT may be specified when the ARROW style is specified.

如果没有特殊说明,子类是可以使用父类的样式的。Button类直接继承自Control类。Control的JavaDoc中关于样式的说明如下所示。

Styles:

BORDER
LEFT_TO_RIGHT, RIGHT_TO_LEFT

Only one of LEFT_TO_RIGHT or RIGHT_TO_LEFT may be specified.



总结起来,可以应用到Button控件上的样式如表3-2所示(同一行中的样式不能同时应用)。不同组的控件可以用“|”组合起来使用,如将SWT.BORDER|SWT.CHECK应用到Button上将创建一个带有边框(BORDER)的复选框(CHECK)。

表3-2 可以在Button上应用的样式

ARROW,CHECK,PUSH,RADIO,TOGGLE
LEFT,RIGHT,CENTER
UP,DOWN,LEFT,RIGHT
BORDER
LEFT_TO_RIGHT,RIGHT_TO_LEFT

即使是同样的样式，应用到不同的控件上，其效果也不是完全相同的。关于某种特定的控件有哪些样式可以应用，以及它们的效果如何，本书将在第4章开始的使用控件的内容中予以介绍。

※ 注意 ※

将同属一组的样式应用到同一个控件上，不会出现编译错误，但运行起来只会有一种样式起作用，读者开发时需要注意这一点。

创建了控件后，还需要设定它在界面上的位置，这部分工作通常由布局管理器完成。在第5章中，将对各种布局管理器作详尽的介绍。

3.2.3 控件的继承检查

在控件类的JavaDoc中，还有一段关于类型继承的说明。如Button类的说明如下：

IMPORTANT: This class is intended to be subclassed only within the SWT implementation.

而Shell类的说明如下：

IMPORTANT: This class is not intended to be subclassed.

这些限制是SWT从扩展性的角度提出的，标识为“不希望被继承”的类型在以后的SWT版本中可能改变其实现方式。如果开发人员不按照这些指示去做的话，SWT不会保证这些程序在以后的SWT版本中仍然可用。

然而JavaDoc只是文本，没有任何限制力。如果不在代码中加以控制的话，这种提示很可能被人忽略，因此SWT使用代码进行运行时的类型检查以提醒开发者。如下代码尝试继承“不希望被继承”的Shell来看SWT是怎样做出这种检查的。

```
public class MyShell extends Shell {
    public MyShell(Display display) {
        super(display);
    }
}
```

创建一个Shell的子类，这个子类什么也没有做，完全继承父类的行为

```
public static void main(String[] args) {
    Display display = Display.getDefault();
    Shell myShell = new MyShell(display);
    myShell.open();
    display.dispose();
}
```

在main函数中创建并打开子类所代表的Shell，因为这段代码会出错，因此也没有编写消息循环



这段代码运行后，出现了一个“不允许子类继承”的SWT异常。

```
Exception in thread "main" org.eclipse.swt.SWTException: Subclassing not
allowed
```

```
at org.eclipse.swt.SWT.error(SWT.java:3374)
at org.eclipse.swt.SWT.error(SWT.java:3297)
at org.eclipse.swt.SWT.error(SWT.java:3268)
at org.eclipse.swt.widgets.Widget.error(Widget.java:435)
at org.eclipse.swt.widgets.Decorations.checkSubclass(Decorations.java:247)
at org.eclipse.swt.widgets.Shell.<init>(Shell.java:267)
at org.eclipse.swt.widgets.Shell.<init>(Shell.java:262)
at org.eclipse.swt.widgets.Shell.<init>(Shell.java:215)
at book.display.ShellExtend$MyShell.<init>(ShellExtend.java:10)
.....
```

根据程序名可以判断，检查就发生在Decorations.checkSubclass这个方法中。该方法实现如下所示。

```
protected void checkSubclass () {
    if (!isValidSubclass () ) error (SWT.ERROR_INVALID_SUBCLASS);
}
```

```
boolean isValidSubclass () {
    return Display.isValidClass (getClass ());
}
```

Display类中的isValidClass方法实现如下所示。

```
static boolean isValidClass (Class clazz) {
    String name = clazz.getName ();
    int index = name.lastIndexOf ('.');
    return name.substring (0, index + 1).equals
(PACKAGE_PREFIX);
}
```



在Display.isValidClass()这个方法中可以看到，Shell的checkSubclass是通过一个类型是否在包org.eclipse.swt.widgets中来判断它是否是一个合法子类。从上面的跟踪也可以发现，只要在子类中重载checkSubclass方法，去掉调用isValidClass检查的逻辑，就可以避开检查，实现对Shell类型的继承了。下面的代码演示了如何创建一个可用的Shell子类(代码见光盘：\book.ch3.MyShell.java)。

```
public MyShell(Display display) {
    super(display);
}

public void checkSubclass() {
}

public static void main(String[] args) {
    Display display = Display.getDefault();
    Shell myShell = new MyShell(display);
    myShell.open();
    display.dispose();
}
```

重载了checkSubclass方法，跳过Shell类的类型检查



这种“绕过”SWT子类检查的方法虽然可以方便一时，但正如SWT的文档所说，在以后的版本中，不保证这些“不标准”的类型可以继续使用，因此少用为宜。

3.2.4 控件的用户数据

控件的用户数据是开发者自定义的一些存储在控件中的属性。这些属性是普通的Java对象，添加或删除属性对控件的使用没有任何影响。可以使用以下方法存取用户数据。

表 3-3 用于访问用户数据的方法

方法名	功能
getData()	读取默认用户数据
setData(Object data)	设置默认用户数据
getData(String key)	以key为标识读取用户数据
setData(String key, Object data)	以key为标识存储用户数据

下面的代码中创建了一个文本框并在其中显示一篇文章的内容。同时，使用用户数据存储了这篇文章的版本和作者。这些信息可以随时从文本框中取出来使用，但是并不会影响文本框的显示内容。

```
Text text = new Text(shell, SWT.NONE);
text.setText("Article Content");
text.setData("Version", "1.2");
text.setData("Author", "Kent Clark");
```



3.2.5 控件的释放

控件也会占用系统资源，这部分资源的分配是在控件的构造函数中完成的。

在之前的示例程序中，都没有编写释放控件资源的代码，这是因为控件是存在于Shell之中的，所以它的父资源是Shell控件，当窗口关闭的时候，所有控件资源都会被释放，因此不需要调用代码去释放控件。

如果在窗口没有关闭之前，强制调用控件的dispose方法去释放它，一般会导致某些问题。下面一段代码对此做了尝试（代码见光盘：\book.ch3.ManualDispose.java）。

```
public static void main(String[] args) {
    Display display = Display.getDefault();
    Shell shell = new Shell(display);
    shell.setSize(100, 100);
    final Label label = new Label(shell, SWT.NONE);
    label.setBounds(10, 10, 20, 20);
    label.setText("A");
    shell.open();
    shell.layout();
    shell.addMouseListener(new MouseAdapter() {
```



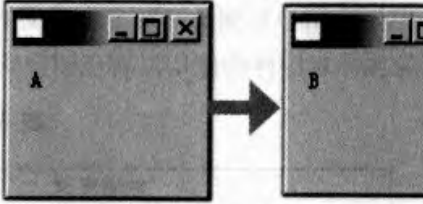
```

        public void mouseDown(MouseEvent event) {
            if (event.button == 1)
                label.dispose();
            else
                label.setText("B");
        }
    };
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
            display.sleep();
    }
}

```

鼠标单击Shell时所触发的事件

这个值为1代表单击左键，2代表单击右键，3代表单击中键



这段代码在窗口里添加了一个标签，并显示字符“A”。在Shell的鼠标事件响应中，添加了如下逻辑，如果单击的是左键，调用Label.dispose将标签释放掉；如果单击的是右键，则将标签的内容改成字符“B”。执行程序后，在窗口中先单击左键后单击右键，会得到一个“控件已被释放”的SWT异常。

当一个控件的dispose方法被调用后，它所占有的系统资源被释放了，但是这个控件所对应的Java对象仍然存在，如果UI线程试图对这个已经被释放的控件进行操作，就会得到“控件已被释放”的异常，因此手动释放控件会导致一系列的问题，如果不是万不得已，最好不要手动释放一个控件，而应该交由SWT系统自动释放它们。

3.3 图形资源

为了使程序界面更加丰富多彩，除了使用控件之外，还需要学会使用其他辅助的图形资源对象。本节将介绍如何使用SWT中几种常见的图形资源，它们都继承自org.eclipse.swt.graphics.Resource。

3.3.1 使用Color

org.eclipse.swt.graphics.Color类管理着操作系统中的颜色资源。Color类使用RGB的色彩模型来描述颜色信息，每一种颜色的数据由分别代表红、绿、蓝三种原色的三个字节（int值0~255）组成。这种模型总共可以描述 2^{24} 种不同的颜色。以下的代码创建并释放了一个代表红色的Color对象。不再需要一个Color对象时，一定要调用它的dispose方法释放占用的系统资源。

```

Color color = new Color(Display.getDefault(), 255, 0, 0);
// 使用颜色
color.dispose();

```

需要一些常用的颜色(黑、白、红等)时，除了自己创建Color实例外，也可以使用Display.getSystemColor方法直接取得。下面的代码得到了一个青色的Color实例。

```

Display display = Display.getDefault();
Color cyanColor = display.getSystemColor(SWT.COLOR_CYAN);

```

※ 注意 : ※

这样取得的Color实例, 在使用完后后不应该调用dispose方法释放它, 因为这些实例可能在其他地方被使用到, 如果在一处释放掉, 其他使用到的地方都会出错, 这也符合前面提到的“谁构造谁释放”的原则, 因为程序只是在使用而没有构造它, 所以使用完后也不应该去释放。



可以使用Color来改变一个控件的前景色和背景色。下面一段代码展示了一个前景色为白色, 背景色为黑色的Label控件(代码见光盘: \book.ch3.ColoredLabel.java)。

```
public static void main(String[] args) {
    Display display = Display.getDefault();
    Shell shell = new Shell(display);
    shell.setSize(120, 80);
    shell.open();

    Color createdWhite = new Color(display, 255, 255, 255);
    Color systemBlack = display.getSystemColor(SWT.COLOR_BLACK);

    Label label = new Label(shell, SWT.NONE);
    label.setBounds(10, 10, 100, 20);
    label.setBackground(systemBlack);
    label.setForeground(createdWhite);
    label.setText("Colored label");
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
            display.sleep();
    }
    createdWhite.dispose();
    display.dispose();
}
```



自己创建一个Color, 从Display中取得一个系统Color, 程序结束时, 对两者的处理是不同的

背景色为黑

前景色为白

每个程序都有的消息循环部分

自己构造的颜色对象要记得释放



3.3.2 使用Image

如果希望设置窗口的图标或在窗口中显示一幅图片, 需要用到Image类, 一个Image对象就代表了一个可用的图片信息。

通常有两种方式来构造一个Image对象, 第一种是使用Image的构造函数指定路径读取一个图像文件或InputStream来构造一个Image, 代码如下所示。

```
Image image = new Image(Display.getDefault(), "C:\\\\graphic.bmp");
```


目前, SWT可以直接读取如下主流图像文件格式。

- ★BMP (Windows or OS/2 Bitmap)
- ★ICO (Windows Icon)
- ★JPEG
- ★GIF
- ★PNG
- ★TIFF

另一种方式是使用ImageData类。ImageData包含着设备无关的图像文件信息, 如图像的尺寸, 每个像素的颜色、透明度等内容。可以通过指定图像的尺寸和每一个像素的颜色值来生成一个ImageData, 代码如下所示。

```
PaletteData palette = new PaletteData(0xFF, 0xFF00, 0xFF0000);  
//设置了RGB三种颜色的掩码  
ImageData imageData = new ImageData(48, 48, 24, palette);  
for (int x = 0; x < 48; x++)  
    for (int y = 0; y < 48; y++)  
        imageData.setPixel(x, y, 0xFF);  
//将48*48的图像全部设置成红色
```

这段代码生成的ImageData代表的图形对象是一个边长为48像素的红色正方形

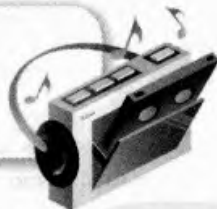
也可以从已有的文件或InputStream构造一个ImageData实例。从图片文件中读取ImageData后, 可以再修改它的内容, 如下面代码所示。

```
ImageData imageData = new ImageData("D:\\test.bmp");
```

创建了ImageData后, 就可以利用它来构造一个Image实例了。不再使用一个Image实例后, 同样要记得释放它。

```
Image image = new Image(Display.getDefault(), imageData);  
//使用Image  
image.dispose();
```

一直在强调的内容: 永远记得释放自己创建的对象



Display内置了几个常用的Image对象, 包括错误图标、警告图标等, 可以用getSystemImage方法访问它们。与使用Color的情况相类似, 使用完这些对象后也不要释放, 如下面代码所示。

```
Display display = Display.getDefault();  
Image image = display.getSystemImage(SWT.ICON_ERROR);
```

Image对象可以用于改变窗口标题栏的图标, 也可以在Label或Button控件上显示图片。下面的程序演示了Image的通常使用方式(代码见光盘: \book.ch3.UsingImage.java)。

```
public class UsingImage {  
    public static void main(String[] args) {  
        Display display = Display.getDefault();  
        Shell shell = new Shell(display);  
        shell.setSize(180, 150);  
        shell.open();  
    }  
}
```

```
Image iconImage = display.getSystemImage(SWT.ICON_QUESTION);
shell.setImage(iconImage);
Image buttonImage = new Image(display, UsingImage.class
    .getResourceAsStream("buttonImage.bmp"));
```

Image 的用法 1: 为窗口设置图标

```
Button button = new Button(shell, SWT.NONE);
button.setBounds(10, 50, 140, 50);
button.setImage(buttonImage);
button.setText("Image Button");
Label label = new Label(shell, SWT.NONE);
label.setImage(buttonImage);
label.setBounds(10, 10, 30, 30);
```

```
while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
        display.sleep();
}
buttonImage.dispose();
display.dispose();
}
```

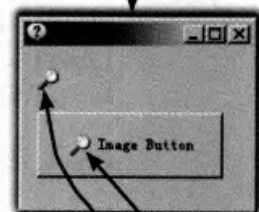


Image 的用法 2: 在控件上显示图像



3.3.3 使用Font

Font资源管理着显示文本时所使用的字体、字号、格式（粗体、斜体等）等各种样式。可以通过指定字体名称、高度和格式创建一个字体，也可以使用Display.getSystemFont得到系统内置的字体。和其他图形资源一样，自己创建的字体用完后一定要释放，而使用系统字体时不能去释放。

可以使用Control.setFont方法设置控件的字体。下面的代码演示了如何创建不同的字体并在Label控件中使用它们（代码见光盘：\book.ch3.UsingFont.java）。

```
public static void main(String[] args) {
    Display display = Display.getDefault();
    Shell shell = new Shell(display);
    shell.setSize(140, 140);
    shell.open();
    Font sysFont = display.getSystemFont();
    Font createdTahoma = new Font(display, "Tahoma", 10, SWT.BOLD);
    Font createdArial = new Font(display, "Arial", 12, SWT.ITALIC);
    Label label1 = new Label(shell, SWT.NONE);
    label1.setBounds(10, 10, 100, 20);
    label1.setText("Tahoma,10,Bold");
    label1.setFont(createdTahoma);
}
```

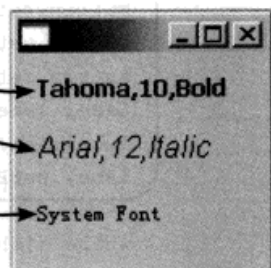


```
Label label2 = new Label(shell, SWT.NONE);
label2.setBounds(10, 40, 100, 20);
label2.setText("Arial,12,Italic");
label2.setFont(createdArial);
```

```
Label label3 = new Label(shell, SWT.NONE);
label3.setBounds(10, 80, 100, 20);
label3.setText("System Font");
label3.setFont(sysFont);
```

```
while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
        display.sleep();
}
```

```
createdTahoma.dispose();
createdArial.dispose();
display.dispose();
}
```



释放自己创建的
那些字体

3.4 高级内容

3.4.1 使用系统托盘

SWT允许Java程序像本地程序一样直接访问系统托盘。在SWT中，系统托盘资源由Tray类型管理，调用Display.getSystemTray方法，可以得到唯一的Tray实例并操作它，如果所在的平台上没有系统托盘，这个方法会返回null。

得到Tray以后，可以通过创建一个TrayItem向系统托盘中添加一个项目，并设置它的图标和提示信息，如下面代码所示（代码见光盘：\book.ch3.UsingSystemTray.java）。

```
public static void main(String[] args) {
    Display display = Display.getDefault();
    Shell shell = new Shell(display);
    shell.setSize(120, 80);
    shell.open();

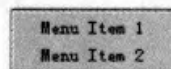
    Tray systemTray = display.getSystemTray();
    TrayItem newItem = new TrayItem(systemTray, SWT.NONE);
    newItem.setImage(display.getSystemImage(SWT.ICON_ERROR));
    newItem.setToolTipText("Test Tray!");
}
```

生成一个TrayItem，设置它的图标和提示文字



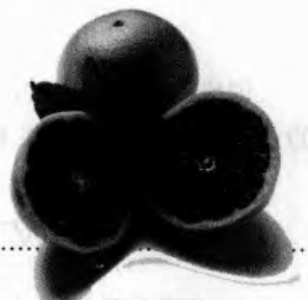
```
final Menu menu = new Menu(shell, SWT.POP_UP);
MenuItem item1 = new MenuItem(menu, SWT.PUSH);
item1.setText("Menu Item 1");
MenuItem item2 = new MenuItem(menu, SWT.PUSH);
item2.setText("Menu Item 2");
```

创建一个有两个菜
单项的菜单



```
newItem.addListener(SWT.MenuDetect, new Listener() {
    public void handleEvent(Event event) {
        menu.setVisible(true);
    }
});
while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
        display.sleep();
}
newItem.dispose();
display.dispose();
}
```

在TrayItem上单击鼠标右键
时, 把菜单显示出来



不再需要系统托盘图标时, 要调用TrayItem.dispose将它释放掉。而Tray对象是不需要释放的。

3.4.2 利用Region构造不规则窗口

Region类型代表的是平面坐标系上的由任意个多边形组成的一个区域。

Region也是一种系统资源。因此使用完后也需要调用dispose方法释放掉。

下面来演示如何创建一个不规则的窗口区域, 代码如下所示(代码见光盘: \book.ch3. UsingRegion.java)。

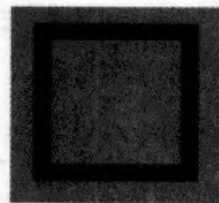
```
public static void main(String[] args) {
    Display display = Display.getDefault();
    final Shell shell = new Shell(display, SWT.NO_TRIM);
```

```
Region region = new Region(display);
region.add(new Rectangle(10, 10, 10, 100));
region.add(new Rectangle(10, 100, 100, 10));
region.add(new Rectangle(10, 10, 100, 10));
region.add(new Rectangle(100, 10, 10, 100));
```

创建一个Region对象, 并向其中添
加了4个矩形区域

```
shell.setRegion(region);
Color color = new Color(null, 255, 0, 0);
shell.setBackground(color);
shell.open();
shell.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseDown(MouseEvent event) {
        shell.close();
    }
});
```

因为窗口没有标题栏, 所以也
没有关闭按钮。因此程序中为
这个Shell添加了一个鼠标事件
监听器: 单击窗口区域时关闭
这个Shell




```

    });
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
            display.sleep();
    }
    region.dispose();
    color.dispose();
    display.dispose();
}

```



创建上面的窗口时，使用了构造函数Shell(Display,int)；使用SWT.NO_TRIM指明了需要创建的是一个没有边框，没有标题栏的窗口。只有这样的窗口才能用setRegion来指定它的形状。

3.4.3 在SWT中使用Swing

在SWT出现之前，已经有众多的Swing程序和控件被开发出来，为了充分利用这些现有资源，SWT需要提供一种在SWT中使用Swing控件的途径。

Swing是通过在AWT的Canvas控件上绘图以显示各种控件；而AWT的控件与SWT的原理相似，都是直接对应到操作系统的控件资源，因此只要用SWT模拟出AWT的Canvas，Swing的控件就可以在其上运行。为此，SWT提供了一个org.eclipse.swt.awt.SWT_AWT类，它通常被称为SWT_AWT桥。

在SWT_AWT中，提供了一个方法用于从SWT的Composite容器得到一个AWT的Frame容器new_Frame(Composite composite)。

有Swing编程经验的读者看到这里应该已经心中有数，既然得到了一个AWT Frame，就可以向这个Frame中添加一个Swing的容器JPanel，之后的工作就和纯Swing编程一般无二了。

下面的代码演示了如何在SWT的Shell中放置一个Swing的Button控件（代码见光盘：\book.ch3.UsingSwing.java）。

```

public static void main(String[] args) {
    Display display = Display.getDefault();
    Shell shell = new Shell(display);
    shell.setBounds(100, 100, 200, 100);
    shell.open();
    Composite composite = new Composite(shell, SWT.EMBEDDED);
    composite.setBounds(0, 0, 200, 100);

    Frame frame = SWT_AWT.new_Frame(composite);
    frame.setLayout(new BorderLayout());
    JPanel panel = new JPanel();
    panel.setLayout(null);
    frame.add(panel, BorderLayout.CENTER);
}

```

SWT_AWT桥在SWT的Composite中生成了一个Swing的容器

```

JButton button = new JButton();
button.setBounds(10, 10, 180, 20);
button.setText("Swing Button");
panel.add(button);

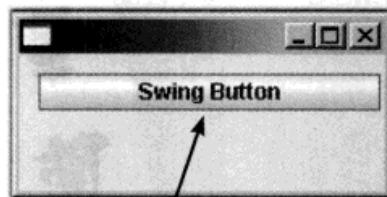
```

创建一个Swing的JButton，并添加到窗口

```

while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
        display.sleep();
}

```



所生成的Swing按钮控件显示在SWT窗口中

3.5 本章小结

学习本章的内容后，读者应该对SWT程序的基本架构，如何使用控件以及使用Image,Font,Color等图形资源的内容有所了解。这些内容是继续学习控件使用的基础，希望读者可以参照书中的示例代码，将每个例子运行一遍以加深理解。从下一章开始，将会对各种常用控件的使用逐一作详细的介绍。

把握好方向，学习起来，那真是容易了很多啊！

第4章

使用基本控件与对话框

拉开崭新的学习帷幕

通过上一章的学习，读者已经掌握了创建一个空白窗口，以及管理和使用字体、图像等图形资源的内容，以这些内容为基础，本章将开始学习部分基本控件的使用。基本控件是构成图形界面的单元，熟练掌握它们是设计更为复杂、强大的界面的基础。通过本章的学习，读者将熟悉在SWT中如何依照不同的需求创建并使用这些基本控件。

本章内容包括：

- ★ Button、Label、Text。
- ★ List。
- ★ Combo。
- ★ ToolBar和ToolBarItem。
- ★ Menu和MenuItem。
- ★ CoolBar和CoolItem。
- ★ TabFolder和TabItem。
- ★ 对话框。

进入第04章

4.1 Button

在GUI界面中,按钮控件Button是使用最为广泛的控件之一。它的主要功能就是对用户的选择动作(鼠标单击控件或当焦点落在控件上时敲空格键)做出反应。下面的代码演示了如何创建并使用不同样式的Button(代码见光盘:\book.ch4.UsingButton.java)。

```

.....
Image image = new Image(display, UsingButton.class
    .getResourceAsStream("demo.gif"));

final Button button = new Button(shell, SWT.NONE);
button.setImage(image);
button.setText("Push Button");
button.setBounds(20, 10, 150, 25);

final Button checkButton = new Button(shell, SWT.CHECK);
checkButton.setImage(image);
checkButton.setText("Check Button");
checkButton.setBounds(20, 45, 150, 20);

final Button radioButton = new Button(shell, SWT.RADIO);
radioButton.setImage(image);
radioButton.setText("Radio Button");
radioButton.setBounds(20, 70, 150, 20);

final Button toggleButton = new Button(shell, SWT.TOGGLE);
toggleButton.setImage(image);
toggleButton.setText("Toggle Button");
toggleButton.setBounds(20, 95, 150, 25);

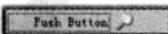
.....

image.dispose();
display.dispose();
    
```

指定控件在界面上的位置
(前两个值是控件左上角的
坐标,后两个值分别代表
控件的宽度和高度)

释放所创建的Image



Button上面显示的文字可以由setText方法设定,而图片则可以由setImage方法设定。默认情况下,图片显示在文字左边,而使用样式SWT.RIGHT_TO_LEFT就可以使图片显示在文字右边,效果如  所示。

上面一段代码中,使用了Button的4种样式。用不同样式创建的Button,除了外形有差异外,被选择后的表现也是不同的。最常见的普通按钮被选择时会显示成凹陷的样子,而选择动作结束后可以自动弹回到没有按过的状态。它被称为Push Button(样式SWT.PUSH)。另外一种Button被按下后不会自动恢复,需要再按一下才能弹起来,这是Toggle Button(样式SWT.TOGGLE)。单选按钮(Radio Button, 样式SWT.RADIO)和复选框(Check Button, 样式SWT.CHECK)也属于Button的家族,

它们会用“加点”和“打勾”的方式来表示自己的选择状态。通过Button.getSelection()方法返回的boolean值,可以判定一个Button当前是否处在选中状态。

※ 注意: ※

*PUSH*样式的Button只有在鼠标|键盘按下时候才能保持选择状态,而放开按键时则立刻恢复到未选状态。而另外三种样式的Button在被选中后都可以保持它们的选中状态。



当用户选择一个Button控件时,会触发一个选择事件。使用SelectionListener中的widgetSelected方法可以监听这个事件并控制程序做出相应的更改。下面的代码演示了如何为Button添加一个选择事件监听器。

```
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        System.out.println("Push Button Selected");
    }
});
```

在上面的代码中,使用了SelectionAdapter类。SelectionAdapter是SelectionListener的空实现,如果只对监听器接口中某一种事件感兴趣,开发者可以继承这个空实现并重载感兴趣的方法而不用实现整个监听器接口来编写所有的方法。对于拥有多个方法的监听器接口,SWT都提供了类似的空实现并以Adapter为名,如MouseListener->MouseAdapter等。

4.2 Label

Label是不能被用户操作的控件,它通常用来在界面上显示图片或不需要修改的文本内容。图4-1中的代码演示了如何使用Label(代码见光盘:\book.ch4.UsingLabel.java),并显示了其运行效果。为了方便观察控件的边界,在示例中用SWT.BORDER样式为Label加上了边框;如果要创建不带边框的Label,可以使用样式SWT.NONE。

```
.....
final Image image1 = new Image(display, Snippet_4_02.class
    .getResourceAsStream("image1.bmp"));
final Image image2 = new Image(display, Snippet_4_02.class
    .getResourceAsStream("image2.bmp"));
final Label label = new Label(shell, SWT.BORDER);
label.setImage(image1);
label.setBounds(10, 10, 120, 50);
final Label textLabel = new Label(shell, SWT.BORDER);
textLabel.setBounds(10, 70, 120, 15);
textLabel.setText("Image 1");
final Button switchButton = new Button
    (shell, SWT.NONE);
```

用Label显示图片

用Label显示文字

这里又是一个使用Button的选择事件监听器的例子



```
switchButton.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(final SelectionEvent e) {
        label.setImage(image2);
        textLabel.setText("Image 2");
    }
});
```

单击按钮时，改变 Label 的图像和文字

```
switchButton.setText("Switch");
switchButton.setBounds(10, 91, 120, 19);
```

.....

```
image1.dispose();
image2.dispose();
display.dispose();
```



初始状态



单击按钮之后



图4-1 使用Label显示图像和文字

与Button类似的，可以使用setText/setImage方法来设置Label中显示的文字和图像。

★ 注意 ★

与Button不同，Label控件不能用来同时显示文本和图像，如果对一个Label先设置文字后设置图像，则Label只会显示图像，反之亦然。

对Label使用样式SWT.SEPARATOR可以使控件显示成一根水平或竖直的分隔线，使用了这个样式后，再对Label使用setImage/setText方法是没有效果的，分隔线的方向需要用样式SWT.HORIZONTAL或SWT.VERTICAL指定，如果没有指定则默认是水平。下面的代码演示了如何使用Label来显示分隔线(代码见光盘：\book.ch4.UsingLabel_2.java)，效果如图4-2所示。

.....

```
Label horLine = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL
| SWT.BORDER);
horLine.setBounds(10, 10, 100, 20);
```

```
Label verLine = new Label(shell, SWT.SEPARATOR | SWT.VERTICAL);
verLine.setBounds(110, 10, 20, 100);
```

.....

SEPARATOR样式的Label会将分隔线画在它的中间位置。分隔线的宽度是无法调节的

使用了BORDER样式的Label控件。使用Label时，出于界面美观的考虑，通常不会使用这种样式

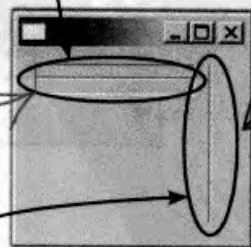


图4-2 使用Label显示分隔线

4.3 Text

需要用户输入信息时，可以使用文本框——Text控件，用户可以在其中输入内容或修改已有的内容。下面的代码创建了一个普通的文本框，它使用了SWT.BORDER样式以显示一个线条边框，如果使用SWT.NONE样式，则将创建一个没有边框的文本框。

```
Text text = new Text(shell, SWT.BORDER);
```

使用getText/setText方法可以得到或改变Text中的文本内容。

如果需要创建一个不可修改（只读）的文本框，可以使用SWT.READONLY样式，这时文本框的背景色将由默认的白色变为灰色，当鼠标单击文字时仍然可以显示编辑光标，也可以拖动鼠标来选择一片文字，对只读的文本框，仍然可以使用setText来改变它的内容。

另外输入一些机密信息（如密码）的时候，用户可能不希望将输入的文本显示在屏幕上，但是又需要知道是否输入过内容。这时可以使用SWT.PASSWORD样式，这个样式会使文本框将输入的所有文字都显示成密码字符（默认为*或●），可以用Text.setEchoChar方法来改变默认密码字符。下面的代码演示了各种不同样式的Text(代码见光盘：\book.ch4.UsingText.java)，界面如图4-3所示。……

```
final Text normalText = new Text(usingTextShell, SWT.BORDER);
normalText.setBounds(10, 10, 112, 20);
normalText.setText("abc");
```

```
final Text passwordText = new Text(usingTextShell, SWT.PASSWORD
| SWT.BORDER);
passwordText.setBounds(255, 10, 112, 20);
passwordText.setText("abc");
```

```
final Text readonlyText = new Text(usingTextShell, SWT.READ_ONLY
| SWT.BORDER);
readonlyText.setBounds(375, 10, 112, 20);
readonlyText.setText("abc");
```

```
final Text borderlessText = new Text(usingTextShell, SWT.NONE);
borderlessText.setText("abc");
borderlessText.setBounds(132, 10, 112, 20);
```

……



图4-3 不同样式的文本框

默认的文本框是单行的，如果需要输入的内容比较多，或者需要换行输入，单行的文本框就无法满足要求了。使用样式SWT.MULTI可以创建一个多行文本框，样式SWT.V_SCROLL和SWT.

H_SCROLL分别控制着在多行文本框中，是否显示竖直滚动条和水平滚动条。

另外一个仅对多行文本框有效的样式是SWT.WRAP，它可以使文本框在一行的文字超过控件宽度时自动换行。如果使用了WRAP样式，H_SCROLL样式将被忽略，水平滚动条不会显示。下面的代码演示了如何使用多行文本框(代码见光盘：\book.ch4.UsingMultiLineText.java)，运行效果如图4-4所示。

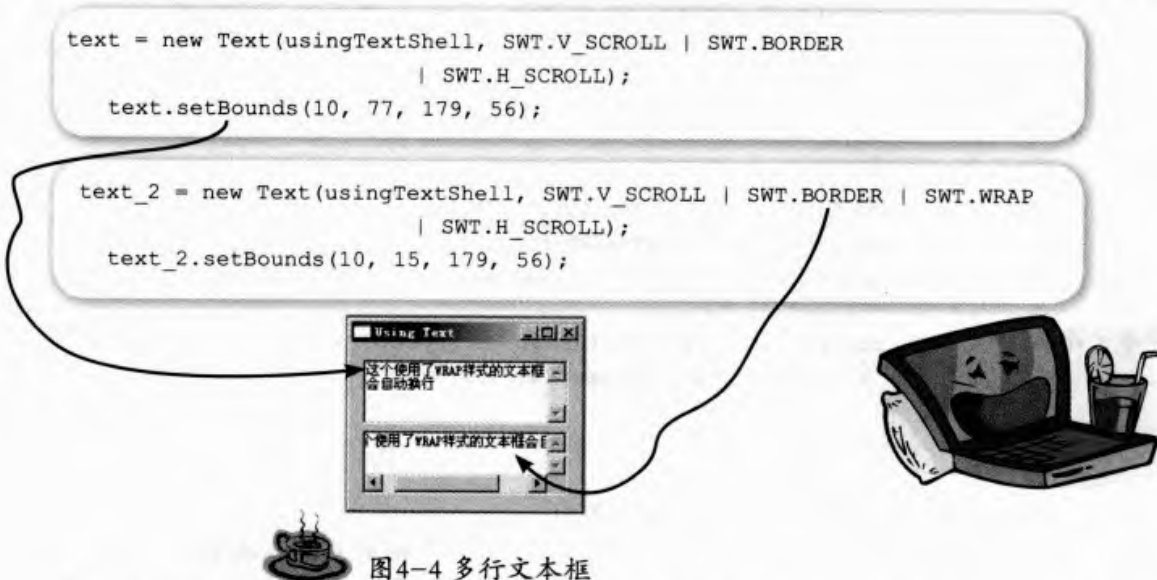


图4-4 多行文本框

Text控件可以响应的事件有DefaultSelection, Modify和Verify。当用户在文本框中输入回车时，会触发DefaultSelection事件，通常这意味着用户已经输入完毕，程序可以利用SelectionListener的widgetDefaultSelected方法监听这个事件并且处理它。Modify和Verify事件都是和Text的内容编辑相关的，当用户编辑Text中的内容时，会发出一个Verify事件，程序可以检查这个事件并判定所作的修改是否可以接受，如果不能接受(在只允许输入数字的Text中输入了字母等)，所做的修改将被拒绝；如果内容修改成功，Text会发出一个Modify事件，通知监听者控件的内容已经改变。图4-5演示了这一流程。

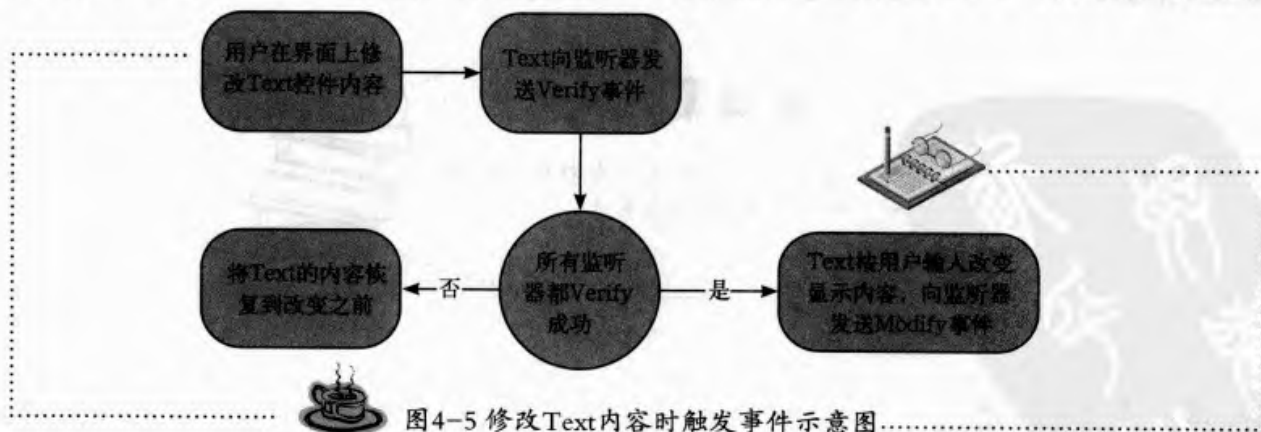


图4-5 修改Text内容时触发事件示意图

Modify事件可以用ModifyListener的modifyText方法监听，而Verify事件则要用VerifyListener的verifyText方法监听。下面的代码演示了如何使用Verify事件监听器创建特殊的文本框。首先是一个只能输入数字的文本框；然后是一个大写文本框，所有输入的小写字母都将转换为大写显示(代码见光盘：\book.ch4.SpecialText.java)。


```

.....
final Text numberText = new Text(shell, SWT.BORDER);
numberText.addVerifyListener(new VerifyListener() {

    public void verifyText(final VerifyEvent e) {
        try {
            Integer.parseInt(e.text);
        } catch (Exception exce) {
            e.doit = false;
        }
    }

});
numberText.setBounds(22, 10, 138, 21);
upperText = new Text(shell, SWT.BORDER);
upperText.addVerifyListener(new VerifyListener() {

    public void verifyText(final VerifyEvent e) {
        e.text = e.text.toUpperCase();
    }

});
upperText.setBounds(22, 40, 138, 20);
shell.open();
.....

```

数字文本框

大写字母文本框

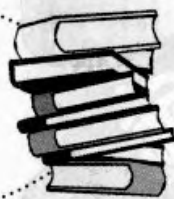
如果新输入的内容不是数字，则将这次修改动作标志为无效

把用户输入的内容变成大写

Verify事件发生时，监听器会收到一个VerifyEvent类型的对象，其中存储了这次改动事件的具体内容。VerifyEvent.text代表了经用户修改后的文字内容，改动这个字段就相当于改动了插入到文本框中的内容；而VerifyEvent.doit是一个有效标志，它允许程序将一次修改事件取消。如果某一个VerifyListener将VerifyEvent.doit设成false，这次修改动作就不会生效，Text的内容也不会被改动。如果所有的VerifyListener都没有改变doit的值，一个Modify事件将会被触发。

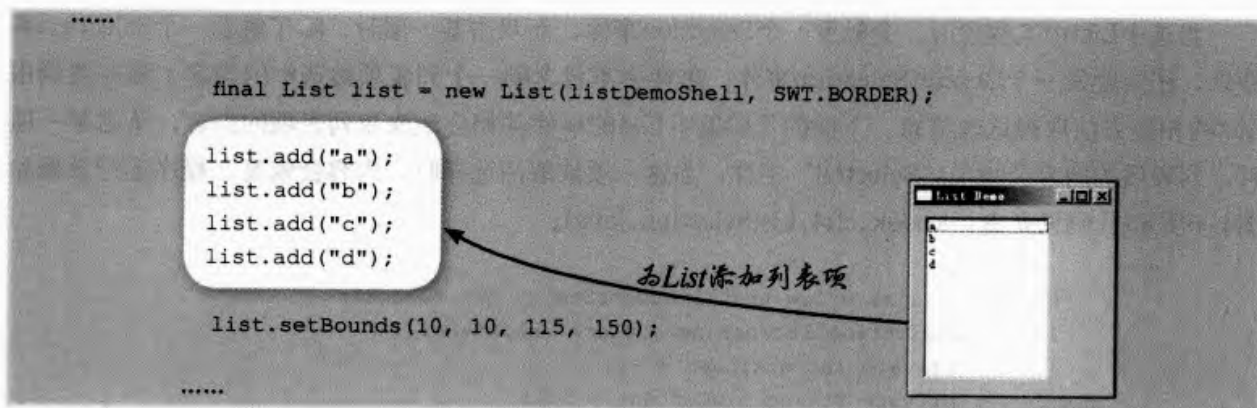
※ 注意：※

使用setText改变文本框中的文字时，同样会触发Verify事件。



4.4 List

List控件用列表的形式向用户展示一组数据，用户可以用鼠标单击其中的一项或多项来做出选择，被选择的项会反白显示。下面的代码创建了一个List并向其中添加了4个列表项(代码见光盘：\book.ch4.UsingList.java)。



List中每一条列表项被称为一个item，表4-1中列出了可用于对item进行操作的方法。

表4-1 用于操作List中item的方法

方法声明	功能
add(String item)	在列表末尾添加一条内容为item的列表项
add(String item, int index)	在由index指定的位置添加内容为item的一项
setItem(int index, String item)	将指定位置的列表项内容设为item
setItems(String[] items)	将列表中所有内容删除，并用items的内容重设列表
remove(int index)	移除指定位置的列表项
remove(int from, int to)	移除从from到to的所有项
remove(int[] indices)	移除indices中指定的所有项
remove(String item)	由前到后移除第一个内容等于item的项
getItem(int index)	返回由index指定的列表项内容
getItems()	返回一个包含所有列表项的String数组
getItemCount()	返回列表项个数

默认情况下，用户只能选择List中的一项；如果希望同时选择多项，可以使用SWT.MULTI样式，这样就可以按住Ctrl键并单击多项以选择它们，也可以按住Shift键后单击两项以选择两者中间的所有项。表4-2列出了与选择项目相关的操作。

表4-2 与选择item有关的操作

操作	功能
select(int index) /deselect(int index)	选择/取消选择由index指定的项
select(int from , int to) /deselect(int from, int to)	选择/取消选择从from到to的所有项
select(int[] indices) /deselect()	选择/取消选择由indices指定的所有项
selectAll()/deselectAll()	选择/取消选择所有内容
getSelection()	得到以一个String数组表示的当前被选择的项
getSelectionCount()	得到当前被选择的项的个数
getSelectionIndex()	得到当前被选择的第一项的index
getSelectionIndices()	得到当前被选择的所有项的index

当选中List中某些项时，会触发一个Selection事件，而双击某一项时，除了触发一个Selection事件外，还会触发一个DefaultSelection事件。事件中不包含哪一个列表项被选中的内容，程序要调用List的相应方法取得这些信息。下面的代码监听List的事件并相应地改变列表项的内容，单击某一项时，列表项的内容会加上“Selected”字样；当这一项被取消选择时，内容会恢复。程序运行效果如图4-6所示(代码见光盘：\book.ch4.ListSelection.java)。

.....

```
final List list = new List(listDemoShell, SWT.BORDER);
list.addSelectionListener(new SelectionListener() {
    private int oldIndex = -1;
    private String oldContent = null;
```

```
    public void widgetSelected(final SelectionEvent e) {
        if (oldIndex != -1)
            list.setItem(oldIndex, oldContent);
        int index = list.getSelectionIndex();
        oldIndex = index;
        oldContent = list.getItem(index);
        list.setItem(index, oldContent + " selected");
    }
    public void widgetDefaultSelected(final SelectionEvent e) {
        if (oldIndex != -1)
            list.setItem(oldIndex, oldContent);
        int index = list.getSelectionIndex();
        oldIndex = index;
        oldContent = list.getItem(index);
        list.setItem(index, oldContent + " default selected");
    }
}
```

当某一项被选中/默认选中时，将对应的项目内容修改掉，不再被选择时则恢复

```
});
list.add("a");
list.add("b");
list.add("c");
list.add("d");
```

.....



没有项被选中时

鼠标单击第一项



选中一项时，其文字会加上selected

鼠标单击第三项



选中另一项时，失去选择的文字会恢复原样



图4-6 监听List的选择事件

4.5 Combo

Combo控件由一个文本框和一个列表组合而成的，当单击文本框右侧的按钮时，会出现一个下拉列表，用户可以在文本框中输入或修改内容，也可以选择列表中预先定义的内容，如图4-7所示。

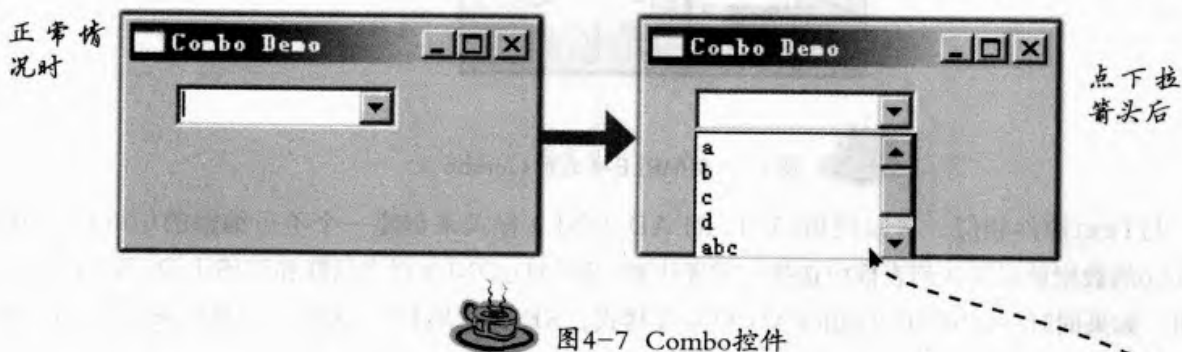


图4-7 Combo控件

Combo的功能与List控件相似，但它比List占用更小的界面空间，因此在界面空间有限的情况下，可以考虑用Combo代替List。下面的代码创建了一个Combo并向其中添加了6个选择项(代码见光盘：\book.ch4.UsingCombo.java)。

```
.....  
Combo combo = new Combo(comboDemoShell, SWT.NONE);  
combo.setItems(new String[] { "a", "b", "c", "d" });  
combo.add("abc");  
combo.add("def");  
.....
```

可以看到，操作Combo中列表项的方法和操作List是完全一样的，但是在选择方面略有不同。Combo无法支持同时选择多项内容(选择的内容要显示在文本框中，选择多项的话无法显示)。因此Combo对选择项的操作只有如表4-3所示的几个方法。

表 4-3 用于选择Combo中的内容的方法

方法	功能
<code>select(int index)</code>	选择第index项
<code>deselect(int index)</code>	取消对第index项的选择。如果当前被选择的项不是index，则没有任何效果
<code>deselectAll()</code>	取消全部选择，文本框被清空

默认样式的Combo会在文本框右面显示一个下拉箭头的Button(参见Button控件的ARROW样式)，单击它会展开列表，这是Combo控件的默认样式SWT.DROP_DOWN。与DROP_DOWN相对的是SWT.SIMPLE样式。使用SIMPLE样式时，下拉箭头的Button不会显示，这时Combo会显示成一个上面是Text，下面是List的控件，这种样式的Combo比较少用。如图4-8所示(代码见光盘：\book.ch4.SimpleCombo.java)。

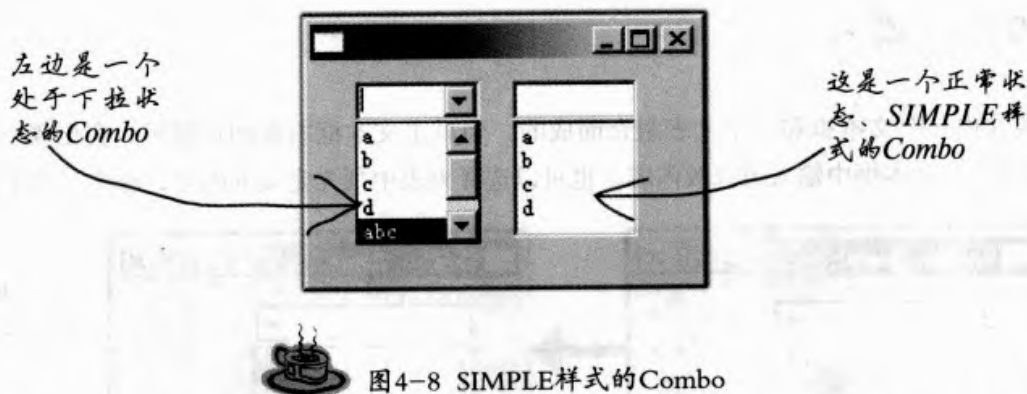


图4-8 SIMPLE样式的Combo

与Text控件相似，可以使用SWT.READ_ONLY样式来创建一个不可编辑的Combo，这时Combo的数据就只能从列表框中选择。需要注意，READ_ONLY样式只能和PUSH_DOWN样式一起使用。如果同时使用SIMPLE和READ_ONLY样式，READ_ONLY会失效，创建出来的Combo仍然是可以编辑的。

从列表框中选择预先定义的内容时，会同时触发Combo的Selection事件和Modify事件；而直接编辑Combo的内容时，只会触发Modify事件。与Text不同，Combo不支持Verify事件，如下面代码所示(代码见光盘：\book.ch4.MonitorCombo.java)。

```
.....

combo = new Combo(shell, SWT.NONE);
combo.setItems(new String[] { "a", "b", "c", "d" });
combo.addModifyListener(new ModifyListener() {
    public void modifyText(ModifyEvent e) {
        Combo combo = (Combo)e.widget;
        String text = combo.getText();
        System.out.println("Combo Text is " + text);
    }
});
combo.setBounds(22, 10, 106, 20);

.....
```

当Modify事件发生时，用ModifyEvent.widget取得被修改的Combo控件，并调用getText方法得到其内容

与Verify事件不同，Modify事件只包含了哪一个控件被修改的信息而不包含有修改的具体内容，程序需要自己从控件中取得修改后的值，可以通过ModifyEvent.widget或ModifyEvent.getSource得到发送事件的控件。

4.6 ToolBar 和 ToolItem

如果想为窗口添加工具栏，需要使用ToolBar控件。图4-9中的代码在窗口中创建了一个有4个按

钮的工具栏(代码见光盘: \book.ch4.UsingToolBar.java), 并显示了其运行效果

```

.....
final ToolBar horToolBar = new ToolBar(shell,
SWT.FLAT | SWT.WRAP);
horToolBar.setBounds(0, 0, 265, 50);

Image toolBarImage1 = new Image(display, UsingToolBar.class
.getResourceAsStream("toolbar1.gif"));
Image toolBarImage2 = new Image(display, UsingToolBar.class
.getResourceAsStream("toolbar2.gif"));
Image toolBarImage3 = new Image(display, UsingToolBar.class
.getResourceAsStream("toolbar3.gif"));
Image toolBarImage4 = new Image(display, UsingToolBar.class
.getResourceAsStream("toolbar4.gif"));

final ToolItem toolItem1 = new ToolItem(horToolBar, SWT.NONE);
toolItem1.setImage(toolBarImage1);
toolItem1.setToolTipText("提示");
toolItem1.setText("Item 1");

final ToolItem toolItem2 = new ToolItem(horToolBar, SWT.PUSH);
toolItem2.setImage(toolBarImage2);
toolItem2.setText("Item 2");

final ToolItem toolItem3 = new ToolItem(horToolBar, SWT.PUSH);
toolItem3.setImage(toolBarImage3);
toolItem3.setText("Item 3");

final ToolItem toolItem4 = new ToolItem(horToolBar, SWT.PUSH);
toolItem4.setImage(toolBarImage4);
toolItem4.setText("Item 4");

.....

toolBarImage1.dispose();
toolBarImage2.dispose();
toolBarImage3.dispose();
toolBarImage4.dispose();
display.dispose();

```

创建一个ToolBar对象

创建用于显示在工具栏按钮上的图像对象

为工具栏按钮设置文字、图像和工具提示信息

在工具栏上创建4个按钮

创建工具栏中的按钮并为它们设置图片、文字等信息

在退出时要释放创建的Image对象





图4-9 创建一个带有图片的工具栏

和其他控件一样, ToolBar可以放置在窗口中的任何位置, 不过习惯上将它放在窗口边缘。可以使用SWT.HORIZONTAL(默认)和SWT.VERTICAL设置工具栏中按钮的排列方向。如图4-10所示是将图4-9中示例代码中创建的工具栏改成竖排的样子。

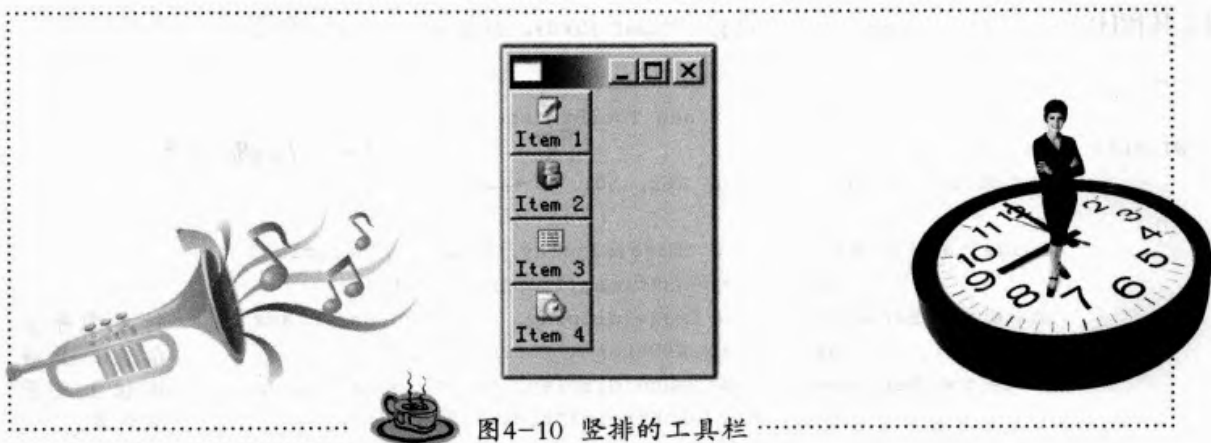


图4-10 竖排的工具栏

工具栏中的按钮由ToolItem控制。下面这段代码在工具栏中创建了一个ToolItem。

```
final ToolItem toolItem1 = new ToolItem(horToolBar, SWT.NONE);
toolItem1.setImage(toolBarImage1);
toolItem1.setToolTipText("提示");
toolItem1.setText("Item 1");
```

与Button相似, ToolItem可以同时显示图片和文本信息, 使用setText/setImage可以设置这些内容。当鼠标指针指在工具栏按钮上并停留一段时间后, 会显示一个浮动窗口提示按钮的用途, 如图4-11所示, 这被称为提示 (ToolTip)。可以使用ToolItem的setToolTipText方法来自定义提示的内容。

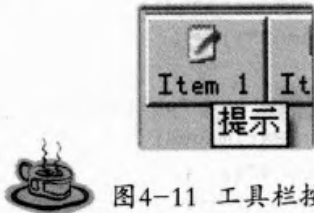


图4-11 工具栏按钮的提示

默认情况下按钮上的文字是显示在图片下方的, 如果对ToolBar使用RIGHT样式, 文字就会显示在按钮右边, 如图4-12所示。

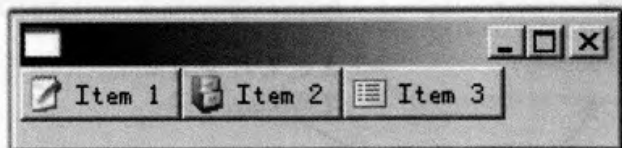
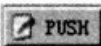
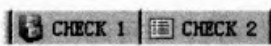

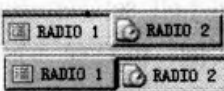
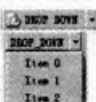


图4-12 使用了RIGHT样式的工具栏按钮

当ToolItem占用的长度超过了ToolBar的长度, 默认情况下越界的ToolItem会无法显示, 这时需要使用WRAP样式。这个样式可以使ToolBar将越界的ToolItem自动移到下一行显示(如果是纵向的工具栏则是移到下一列)。

表4-4列出了ToolItem可以使用的样式及其效果。

表 4-4 ToolItem 的样式

样式	效果	说明
PUSH		最普通的样式，ToolItem 会显示成一个使用了 PUSH 样式的 Button
CHECK		显示成 TOGGLE 样式的 Button。虽然样式是 CHECK 但并不会显示一个 CHECK Button 的可以打勾的框出来
RADIO		同样显示成 TOGGLE 样式的 Button。与 CHECK 的不同在于同一个 ToolBar 中使用了 RADIO 样式的 Item 只有一个能被选中，选中了一个以后，其他的自动恢复到未选中状态
SEPARATOR		显示成一条分隔线，可以用来隔开并分组其他 ToolItem。分隔线的方向会根据工具栏的方向自动调整
DROP_DOWN		这种样式的按钮旁边会有一个下拉箭头，单击后可以打开一个菜单，在其中可以添加其他命令。关于使用菜单的内容，请参照第 4.7 节 Menu 和 MenuItem 的使用

ToolItem 仅支持 Selection 一种事件。当工具栏按钮被单击时，对应的 ToolItem 触发一个 Selection 事件，如下面代码所示，程序响应这个事件就可以实现工具栏操作。

```
toolItem1.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(final SelectionEvent e) {
        //处理事件
    }
});
```

如果普通的工具栏按钮不能满足需求，也可以将其他控件，如 Text、Combo 等放置在工具栏上。下面的代码演示了如何在工具栏上创建一个 Text (代码见光盘：\book.ch4.EnhancedText.java)。

```
.....
final ToolBar horToolBar = new ToolBar(shell, SWT.SHADOW_OUT | SWT.RIGHT);
horToolBar.setBounds(0, 0, 500, 50);

final ToolItem toolItem1 = new ToolItem(horToolBar, SWT.SEPARATOR);
Text text = new Text(horToolBar, SWT.BORDER);
text.setSize(120, 20);
toolItem1.setControl(text);
toolItem1.setWidth(120);
.....
```

需要调整 ToolItem 的宽度使
控件能够显示



在工具栏上创建其他
控件时，首先要创建
一个 SEPAARATOR 样
式的 ToolItem 来摆放这
个控件

4.7 Menu和MenuItem

一个好的GUI程序少不了菜单的鼎力支持。通常所见的菜单有两种形式，一种是单击窗口中的菜单栏出现的下拉式菜单；另一种是在界面上单击右键出现的弹出式菜单。

为一个窗口添加菜单栏时，需要使用Menu的BAR样式。与可以任意摆放的ToolBar不同，菜单栏只能处于窗口顶部，无法调整其位置，也没有什么样式可以自定义显示风格。下面的代码在窗口中创建了一个菜单栏并将它显示在界面上。

```
Menu bar = new Menu(shell, SWT.BAR);
shell.setMenuBar(bar);
```

创建Menu的语法和其他控件没有区别，但是多了一句shell.setMenuBar。SWT是不允许在一个窗口中显示多个菜单栏的，程序可以创建任意多个Menu实例，但只有一个可以作为窗口的菜单栏显示，因此需要用Shell.setMenuBar来决定哪一个实例可以显示在Shell中。

有了菜单栏后，就要向其中添加下拉菜单了。一个下拉菜单由一个使用了CASCADE样式的MenuItem实例和一个Menu实例组成。MenuItem负责在菜单栏上显示一个可单击的位置，当用户选择这个位置时显示对应的下拉菜单；而Menu则控制着下拉菜单的具体内容。向下拉菜单中添加菜单项时，需要使用MenuItem的PUSH样式，这是MenuItem的默认样式。下面的代码演示了如何创建菜单栏和子菜单(代码见光盘：\book.ch4.UsingMenu.java)，程序运行效果如图4-13所示。

```
.....
Image menu1 = new Image(Display.getDefault(),
Snippet_4_07_1.class.getResourceAsStream("menu1.gif"));
Image menu2 = new Image(Display.getDefault(),
Snippet_4_07_1.class.getResourceAsStream("menu2.gif"));
```

```
Menu bar = new Menu(shell, SWT.BAR);
shell.setMenuBar(bar);
```

代表菜单栏的Menu对象

```
MenuItem fileMenuItem = new MenuItem(bar, SWT.NONE);
fileMenuItem.setImage(menu1);
fileMenuItem.setText("File");
```

在菜单栏上创建一个菜单项

```
MenuItem otherMenuItem = new MenuItem(bar, SWT.PUSH);
otherMenuItem.setText("Other");
```

```
final Menu fileMenu = new Menu(fileMenuItem);
fileMenuItem.setMenu(fileMenu);
```

第二个菜单项

创建一个新的menu并将它和菜单项关联起来，这样单击菜单项就会显示这个菜单

```
final MenuItem openMenuItem = new MenuItem(fileMenu, SWT.PUSH);
openMenuItem.setText("Open");
openMenuItem.setImage(menu1);
```

向下拉菜单中添加菜单项

```
final MenuItem exitMenuItem = new MenuItem(fileMenu, SWT.NONE);
exitMenuItem.setText("Exit");
exitMenuItem.setImage(menu2);
```

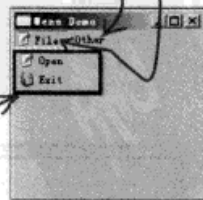


图4-13 创建一个简单的菜单栏

图4-14显示了创建菜单的层级结构。

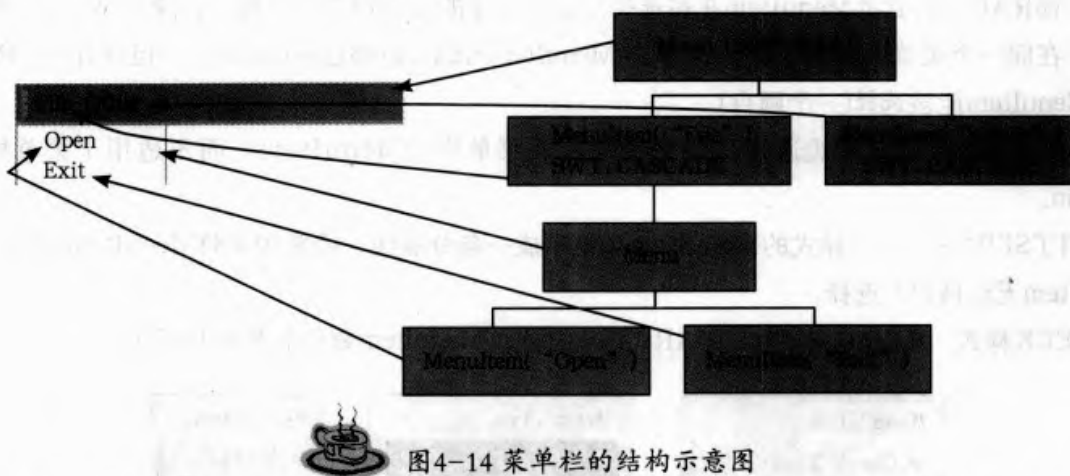


图4-14 菜单栏的结构示意图

上面代码中首先生成了一个BAR样式的Menu，然后向其中添加了两个MenuItem，又将fileMenu关联到了第一个MenuItem上面，这样就做成了下拉菜单的样式，而之后向fileMenu中添加的MenuItem，就作为下拉菜单的菜单项显示出来，如果再将一个Menu关联到下拉菜单中的MenuItem，就会形成子菜单的样式。将上面的代码改造如下，效果如图4-15所示。

```
final MenuItem openMenuItem = new MenuItem(fileMenu, SWT.CASCADE);
openMenuItem.setText("Open");
openMenuItem.setImage(menu1);
```

把需要添加子菜单的菜单项样式改成 SWT.CASCADE

```
final Menu openMenu = new Menu(openMenuItem);
openMenuItem.setMenu(openMenu);

final MenuItem openDirMenuItem = new MenuItem(openMenu, SWT.PUSH);
openDirMenuItem.setText("Directory");

final MenuItem openFileMenuItem = new MenuItem(openMenu, SWT.PUSH);
openFileMenuItem.setText("File");
```

生成一个包含两个菜单项的菜单并将它关联到上一层的MenuItem上面



图4-15 层叠Menu

按照上面代码中的方法反复，可以构成任意多层次的Menu。

上面的代码中用到了MenuItem的两种样式，CASCADE和PUSH，在特殊需要下，还可以对MenuItem使用另外三种样式，CHECK,RADIO和SEPARATOR。

使用了CHECK样式的MenuItem在被选择后会在文字的左边加上一个勾，再次选择这个item会将勾去掉；而RADIO样式的MenuItem在被选择后会在文字的左边加上一个圆点；除此之外，它们的区别还在于在同一个菜单项中，CHECK样式的MenuItem可以同时被选中(多个勾)，但只有一个RADIO样式的MenuItem能被选择(一个圆点)。

CHECK和RADIO样式通常只适用于下拉式菜单中的MenuItem，而不适用于菜单栏中的MenuItem。

使用了SEPARATOR样式的MenuItem会显示成一条分隔线，通常用来将MenuItem分组。这样的MenuItem无法被用户选择。

CHECK样式、RADIO样式和SEPARATOR样式的MenuItem效果如图4-16所示。



图4-16 CHECK样式、RADIO样式和SEPARATOR样式的MenuItem

为了响应用户选择一个菜单项的操作，需要对MenuItem的Selection事件做监听，另外，MenuItem还支持Arm和Help两种事件。当用户将焦点移动到菜单项上面时(菜单项反白显示时)，会触发一个Arm事件，可以使用这个事件为菜单制造一些动画效果；而如果这时用户按帮助键(在Windows系统中是F1键)，则会触发一个Help事件。下面的代码演示了如何使用这些事件(代码见光盘：\book.ch4.MonitorMenu.java)。

.....

```
Menu bar = new Menu(shell, SWT.BAR);
shell.setMenuBar(bar);
MenuItem fileMenuItem = new MenuItem(bar, SWT.CASCADE);
fileMenuItem.setText("File");
final Menu fileMenu = new Menu(fileMenuItem);
fileMenuItem.setMenu(fileMenu);
final MenuItem openMenuItem = new MenuItem(fileMenu,
SWT.PUSH);
```

```
openMenuItem.addHelpListener(new HelpListener() {
    public void helpRequested(final HelpEvent e) {
        MessageBox msgBox = new MessageBox(shell, SWT.ICON_INFORMATION
            | SWT.OK);
        msgBox.setText("Help Message");
        msgBox.setMessage("Choose this to open a new file");
        msgBox.open();
    }
});
```

```
openMenuItem.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(final SelectionEvent e) {
        FileDialog fd = new FileDialog(shell, SWT.OPEN);
```

打开一个消息框，显示一行说明文字



```

        fd.open();
    }
    });
    openMenuItem.setText("Open");
    final MenuItem exitMenuItem = new MenuItem(fileMenu, SWT.NONE);
    exitMenuItem.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(final SelectionEvent e) {
            shell.close();
        }
    });
    exitMenuItem.setText("Exit");
    .....

```

单击Exit菜单项时关闭窗口



当用户将鼠标移动到Open菜单上并按F1键时，会显示一个消息对话框。关于消息对话框的使用内容，将在本章后面的部分予以介绍。

4.8 CoolBar和CoolItem

另外一种可以用来创建工具栏的控件就是CoolBar。与ToolBar不同，CoolBar中的所有按钮都是可以拖动以改变位置的，用户可以按照喜好自行安排工具栏中按钮的位置。

ToolBar上的按钮是由ToolItem控制的，而CoolItem却只是作为一个可以拖动的容器而存在，开发者要自己向CoolItem中加入Button以创建一个工具栏按钮。作为容器，CoolItem不仅可以容纳Button，也可以容纳Text或Combo控件，这意味着可以将文本框和组合框使用在工具栏中，创建出比ToolBar更为强大的工具栏。下面的代码演示了如何使用CoolBar创建一个与前面讲述ToolBar时所创建的工具栏类似的工具栏(代码见光盘：\book.ch4.UsingCoolBar.java)。

.....

```

final CoolBar coolBar = new CoolBar(shell, SWT.NONE);
coolBar.setBounds(0, 0, 400, 30);

```

创建CoolBar控件

创建一个CoolItem并利用setControl方法向其中添加一个Button



```

final CoolItem coolItem1 = new CoolItem(coolBar, SWT.PUSH);
final Button button1 = new Button(coolBar, SWT.NONE);
button1.setImage(toolBarImage1);
button1.setSize(25, 25);
coolItem1.setControl(button1);
coolItem1.setSize(coolItem1.computeSize(25, 25));

```

使用setSize调整CoolItem的尺寸，尺寸不对会导致其中包含的控件无法显示



```

final CoolItem coolItem2 = new CoolItem(coolBar, SWT.PUSH);

```



```
final Button button2 = new Button(coolBar, SWT.NONE);
button2.setImage(toolBarImage2);
button2.setSize(25, 25);
coolItem2.setControl(button2);
coolItem2.setSize(coolItem2.computeSize(25, 25));
```

创建另外几个CoolItem

```
final CoolItem coolItem3 = new CoolItem(coolBar, SWT.PUSH);
final Button button3 = new Button(coolBar, SWT.NONE);
button3.setImage(toolBarImage3);
button3.setSize(25, 25);
coolItem3.setControl(button3);
coolItem3.setSize(coolItem3.computeSize(25, 25));
```

```
final CoolItem coolItem4 = new CoolItem(coolBar, SWT.PUSH);
final Button button4 = new Button(coolBar, SWT.NONE);
button4.setImage(toolBarImage4);
button4.setSize(25, 25);
coolItem4.setControl(button4);
coolItem4.setSize(coolItem4.computeSize(25, 25));
```

.....

```
toolBarImage1.dispose();
toolBarImage2.dispose();
toolBarImage3.dispose();
toolBarImage4.dispose();
display.dispose();
```

释放程序中使用
的图像资源

代码中首先创建了一个CoolBar，然后向其中放置了4个CoolItem，又在每个CoolItem中各放置了一个Button来显示工具栏的按钮图标，CoolItem使用setControl方法来设定它里面包含的控件。与ToolItem不同，CoolItem本身不触发任何事件，事件是交由其中所包含的Button自行管理的。

理论上讲，所有控件都可以放在CoolItem中显示，然而某些占用空间过大的控件，如List等是不适合放置在其中的。下面的代码将一个Combo放在了作为工具栏的CoolBar中(代码见光盘：\book.ch4.ComboCoolBar.java)，效果如图4-17所示。

```
final CoolBar coolBar = new CoolBar(shell, SWT.NONE);
coolBar.setBounds(0, 0, 400, 30);
```

```
final CoolItem coolItem1 = new CoolItem(coolBar, SWT.PUSH);
final Combo combo = new Combo(coolBar, SWT.NONE);
combo.setSize(80, 25);
coolItem1.setControl(combo);
coolItem1.setSize(coolItem1.computeSize(80, 25));
```

除Combo外，Text等
尺寸合适的控件都可
以放在CoolItem中



图4-17 在CoolBar中放置Combo

工具栏可以拖动是个好功能，但有时为防止用户的误操作，可能希望禁止它。可以使用 `CoolBar.setLocked(true)` 来禁止 `CoolBar` 中的 `item` 被重新排列。

在上面的例子中，为每个工具栏按钮安排了一个 `CoolItem`，这样它们可以被任意排列，但是如果希望将某几个按钮固定地分在一组中，`CoolBar` 就无能为力了，而且如果读者做过尝试的话，会发现 `CoolItem` 中的 `Button` 是不能像 `ToolItem` 或普通的 `Button` 控件一样同时显示图片和文字的，这些问题都可以很容易地通过将 `ToolBar` 放到 `CoolBar` 里面而解决。下面的代码演示了如何在 `CoolBar` 中创建 `ToolBar` (代码见光盘：\book.ch4.ToolBarCoolBar.java)，效果如图4-18所示。

```

.....

final CoolBar coolBar = new CoolBar(shell, SWT.NONE);
coolBar.setBounds(0, 0, 400, 30);
final CoolItem coolItem1 = new CoolItem(coolBar, SWT.PUSH);

final ToolBar toolBar1 = new ToolBar(coolBar, SWT.NONE);
ToolItem item10 = new ToolItem(toolBar1, SWT.NONE);
item10.setImage(toolBarImage1);
ToolItem item11 = new ToolItem(toolBar1, SWT.NONE);
item11.setImage(toolBarImage2);
coolItem1.setControl(toolBar1);

toolBar1.pack();
coolItem1.setSize(coolItem1.computeSize(
    toolBar1.getSize().x, toolBar1.getSize().y));

final CoolItem newItemCoolItem = new CoolItem(coolBar,
    SWT.PUSH);
.....
    
```

在 `CoolBar` 中放置 `ToolBar` 的步骤与放置其他控件并无区别，创建控件后利用 `setControl` 方法将它关联到 `CoolItem` 上

使用 `Control.pack()` 方法使 `ToolBar` 计算其自身尺寸后，得到它的尺寸用来设置 `CoolItem` 的尺寸

包含在同一个 `CoolItem` 中的工具栏按钮顺序固定

可以任意拖动 `CoolItem` 改变按钮组的位置



图4-18 将 `CoolBar` 和 `ToolBar` 组合起来

在每一个 `CoolItem` 中放置一个 `ToolBar`，再将需要分组的按钮作为同一个 `ToolBar` 的 `ToolItem`，就可以实现既能够让用户拖动，又可以保持按钮分组不变的强大工具栏。Eclipse IDE 中就普遍使用了这种工具栏。

4.9 TabFolder 和 TabItem

使用 `TabFolder` 可以将多个页面层叠排列，并为每一个页面显示一个选项卡，当用户单击这些选项卡时，可以切换位于最上层的页面，`TabFolder` 中每一个选项卡由一个 `TabItem` 管理。`TabFolder` 的选项卡默认是处于上端的，如图4-19所示。可以使用 `SWT.BOTTOM` 样式使它显示在下端。

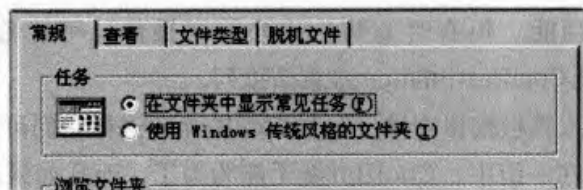


图 4-19 TabFolder示例

在选项卡上面，也是可以显示文字和图片的。下面的代码在窗口中创建了一个TabFolder并在其中添加了两个选项卡(代码见光盘：\book.ch4.UsingTab.java)，效果如图4-20所示。

.....

```
final TabFolder tabFolder = new
TabFolder(tabfolderDemoShell, SWT.NONE);
tabFolder.setBounds(10, 10, 242, 234);
```

```
final TabItem tab1TabItem = new TabItem(tabFolder,
SWT.NONE);
tab1TabItem.setImage(image1);
tab1TabItem.setText("Tab 1");
```

```
final Composite composite_1 = new Composite(tabFolder,
SWT.NONE);
tab1TabItem.setControl(composite_1);
```

```
final TabItem tab2TabItem = new TabItem(tabFolder,
SWT.NONE);
tab2TabItem.setText("Tab 2");
tab2TabItem.setImage(image2);
```

.....

创建一个TabItem来生成一个选项卡，并设置它的文字和图像

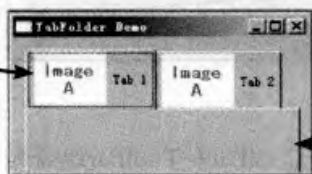


图4-20 创建一个TabFolder

创建一个TabFolder容器用来放置选项卡

创建一个Composite用来在选项卡中放置选项页面的内容，用setControl方法将它和选项卡关联起来

与上一节讲到的CoolItem类似，TabItem是通过setControl方法来设置其中显示的控件的。这个方法不能接受多个控件作为参数，而程序通常需要在TabItem中放置很多内容。这时，就需要先在TabItem中放一个Composite控件，然后再在Composite中放置其他控件。Composite这种专门用来容纳其他控件的控件被称为容器。下一章将专门介绍有关容器使用的内容。

当用户切换选项卡的时候，TabFolder会发送一个Selection事件。但与处理List时的情况类似，这个事件并不包含哪一个TabItem被选中的信息。可以用TabFolder的getSelectionIndex或getSelections方法得到被选中的TabItem并做相应的处理，不过通常情况下开发者不需要关心这个事件。下面的代码处理TabFolder的选择事件(代码见光盘：\book.ch4.MonitorTab.java)。

```
tabFolder.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(final SelectionEvent e) {
        int index = tabFolder.getSelectionIndex();
        TabItem item = tabFolder.getItem(index);

        System.out.println("The tab with name " + item.getText()
            + " is selected");
    }
});
```

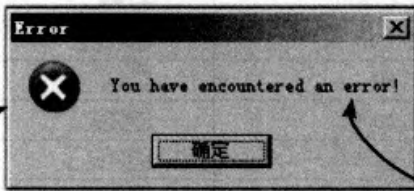
打印出用户选择的TabItem的名字

4.10 对话框

操作系统为许多通用信息提供了标准的对话框。本节中会介绍如何在SWT中调用这些对话框。

4.10.1 消息框

如果程序出了错，开发者可以使用任何一种他喜欢的方式来通知用户，但是使用系统提供的错误对话框(显示一个红叉的图标)显得更为专业，同样，在有信息需要显示或要求用户对某个问题做“是/否”回答时，都可以使用系统提供的标准消息对话框。如图4-21所示是Windows系统提供的错误对话框(代码见光盘：\book.ch4.UsingMessageBox_1.java)。



设置对话框的标题

设置对话框的文字

图4-21 一个错误对话框

SWT提供了MessageBox类，使用这个类型可以方便地生成标准对话框。下面的代码演示了如何创建上面的错误对话框。

```

    创建一个MessageBox对象
    MessageBox mb = new MessageBox(parentShell, SWT.ICON_ERROR | SWT.OK);

    mb.setText("Error");

    mb.setMessage("You have encountered an error!");






    mb.open();  // 显示对话框
    
```



创建MessageBox时需要指定一个父Shell和使用的样式。使用setText来设定MessageBox的标题, setMessage来设定要显示的消息, 其他的部分则交给样式来定义。MessageBox的样式由图标和按钮两部分组成。

图标样式决定着MessageBox中显示的图标。可用的图标如表4-5所示(以Windows XP系统为例)。

表 4-5 可用的对话框图标样式

样式名	说明	图标
SWT.ICON_ERROR	显示错误信息	
SWT.ICON_WARNING	显示警告信息	
SWT.ICON_INFORMATION	显示提示信息	
SWT.ICON_WORKING	显示正在进行的工作信息	
SWT.ICON_QUESTION	显示问题	

如何使用这些图标是程序员的自由, 即使在显示错误信息的时候使用一个问号图标也是可以的。但是一个消息框只能使用一种图标, 即使使用了多个, 也只有一个会起作用。









按钮的样式决定着在消息框中显示哪些按钮。可用的按钮如表4-6所示。

表 4-6 可用的按钮样式

样式名	效果
SWT.OK	
SWT.CANCEL	
SWT.YES	
SWT.NO	
SWT.ABORT	
SWT.RETRY	
SWT.IGNORE	

而可使用的按钮组合如表4-7所示。

表 4-7 对话框中可用的按钮组合

组合名	效果
SWT.OK SWT.CANCEL	 
SWT.YES SWT.NO SWT.CANCEL	  
SWT.ABORT SWT.RETRY SWT.IGNORE	  

在显示对话框时可以任意选择一个按钮组合, 也可以使用组合中的某个子集, 如SWT.RETRY | SWT.IGNORE, 但是如果试图使用不同组合中的按钮, 消息框就会只显示一个OK按钮。

无论用户在消息框中单击了什么按钮, 消息框都会关闭。在打开消息框时, 调用了MessageBox.open方法, 这个方法的返回值是一个整型。它标识了被单击的按钮信息, 使用这个信息就可以得知用户单击了什么按钮。下面的代码演示了如何得到这个信息并处理它(代码见光盘: \book.ch4. UsingMessageBox_2.java)。

.....

```
final Shell messageboxDemoShell = new Shell(display);
messageboxDemoShell.setText("MessageBox Demo");
messageboxDemoShell.setSize(224, 79);
messageboxDemoShell.open();
```

创建一个Shell

```
Button showMessageButton = new
Button(messageboxDemoShell, SWT.PUSH);
showMessageButton.setText("Show the Message Box");
showMessageButton.setBounds(10, 10, 196, 24);
showMessageButton.addSelectionListener(new
```

在Shell中放置一个Button，单击它就会打开对话框

```
SelectionAdapter() {
```

```
    public void widgetSelected(SelectionEvent e) {
        MessageBox mb = new
        MessageBox(messageboxDemoShell, SWT.ICON_QUESTION | SWT.OK |
        SWT.CANCEL);
        mb.setText("Choose");
        mb.setMessage("Which Button will you choose?");
```

```
        MessageBox resBox = new
        MessageBox(messageboxDemoShell,
```

```
                SWT.ICON_INFORMATION | SWT.OK);
        resBox.setText("The result");
```

```
        int result = mb.open();
```

根据open方法的返回值判定用户单击的按钮

```
        if (SWT.OK == result) { // 如果用户单击的是OK
            resBox.setMessage("You have chosen OK button!");
        } else if (SWT.CANCEL == result) { // 用户单击了Cancel或窗口的关闭按钮
            resBox.setMessage("You have chosen Cancel Button!");
        }
        resBox.open();
```

```
    });
    .....
}
```

当用户单击某一按钮时，open方法的返回值就是样式中代表该按钮的值

4.10.2 文件与目录对话框

需要让用户选择文件或目录时，可以使用文件与目录对话框，如图4-22所示。

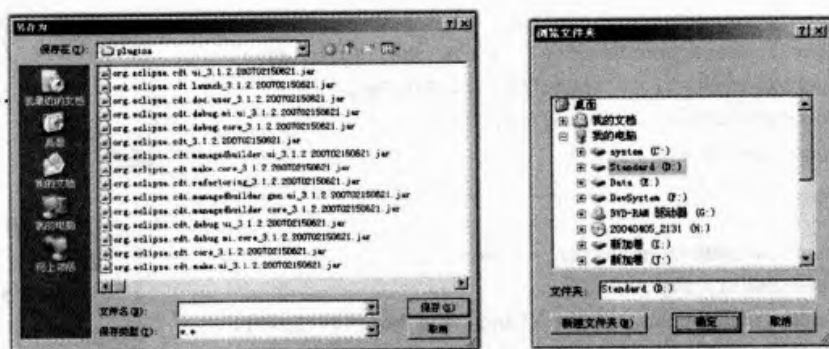


图 4-22 文件对话框(左)与目录对话框(右)

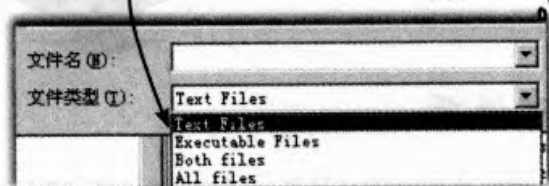
文件对话框对应的SWT类型是FileDialog。创建FileDialog时，有三种样式可以选择，SAVE, OPEN和MULTI。选择SAVE样式时，对话框的标题会显示为“另存为”；而选择OPEN/MULTI样式时，标题则是“打开”；如果选择了MULTI样式，可以在对话框中选择多个文件，而OPEN样式时则只能选择一个文件。下面的代码创建了一个SAVE对话框并打开它(代码见光盘：\book.ch4.FileDialog_1.java)。

```
FileDialog fileDialog = new FileDialog(shell, SWT.SAVE);
fileDialog.open();
```

FileDialog.setFileName可以用来设定文件对话框打开时默认的文件名。比如可以为SAVE对话框中的文件默认命名为“Default”或其他什么内容。

在文件对话框中，有一个“文件类型”的下拉框，只有扩展名与选择的类型相同的文件才会显示在对话框中，这被称为文件对话框的过滤器(Filter)，使用FileDialog的两个方法setFilterExtensions和setFilterNames可以为文件对话框配置过滤器，代码及效果如下所示(代码见光盘：\book.ch4.FileDialog_2.java)。

```
fileDialog.setFilterNames(new String[] {
    "Text Files",
    "Executable Files",
    "Both files",
    "All files" });
fileDialog.setFilterExtensions(new String[] {
    "*.txt",
    "*.exe",
    "*.exe;*.txt",
    "*.*" });
```



两个数组中的项是
一一对应的

可以看出，两个数组分别声明了在“文件类型”组合框中显示的内容和它所对应的文件扩展名。可以用分号分隔多个扩展名，并且使用“*”通配符来表示任意文字。

默认情况下FileDialog在打开时会把目录自动设定为系统目录。自定义打开时的目录需要使用FileDialog.setFilterPath方法设定。下面的代码将默认目录设为D盘的根目录。

```
fileDialog.setFilterPath("D:\\");
```

为了得到用户选择的文件信息，可以使用FileDialog.getFileNames方法，它会将用户选择的文件相对于filterPath的路径以一个String数组的形式返回；另外FileDialog.open/FileDialog.getFileName方法也会返回一个代表文件绝对路径的String，但是它们只适用于对话框样式非MULTI样式的情况，如果选择了多个文件，调用这两个方法只会返回第一个文件的路径。如果用户在对话框中没有选择确定而是选择了取消，上面的方法都只会返回null。下面的代码演示了如何从FileDialog中取得文件路径(代码见光盘：\book.ch4.FileDialog_3.java)。

```
FileDialog fileDialog = new FileDialog(shell, SWT.MULTI);
if (null != fileDialog.open()) {
    System.out.println("You have chosen the following files :");
    String[] files = fileDialog.getFileNames();
    for (int i = 0; i < files.length; i++)
        System.out.println(fileDialog.getFilterPath() + files[i]);
}
```

目录对话框所对应的SWT类是DirectoryDialog。DirectoryDialog没有可以自定义的样式，它的操作相对来说也简单一些，创建一个DirectoryDialog后，调用其open方法可以打开对话框，open返回的值就是所选择的目录的绝对路径，使用setFilterPath可以设定对话框的初始目录。下面的代码演示了如何使用一个DirectoryDialog(代码见光盘：\book.ch4.UsingDirectoryDialog.java)。

```
DirectoryDialog dirDialog = new DirectoryDialog(shell);
```

← 创建一个目录对话框

```
dirDialog.setFilterPath("D:\\");
```

← 将初始目录设置为D盘根目录

```
System.out.println(dirDialog.open());
```

← 打开对话框并打印
用户选择的目录

4.10.3 颜色对话框

颜色对话框 (ColorDialog) 显示一个调色板并允许用户在其中选择一种颜色。如图4-23所示。

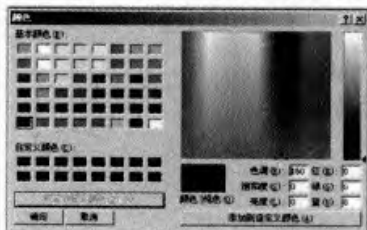


图4-23 颜色对话框

ColorDialog的open方法会返回一个RGB实例，代表了用户选择的颜色类型。下面的代码演示了如何显示一个颜色对话框并获得用户选择的颜色(代码见光盘：\book.ch4.UsingColorDialog.java)。

```
ColorDialog cd = new ColorDialog(shell);
RGB rgb = cd.open();
```

创建一个颜色对话框并打开它，如果用户选择了颜色并单击“确定”按钮，open方法会返回一个RGB对象，否则则返回null

```
if(rgb != null) {
    Color color = new Color(Display.getDefault(),rgb);
    //Use Color
    color.dispose();
}
```

使用返回的RGB对象创建Color对象并在程序中使用

4.10.4 字体对话框

字体对话框列出当前系统中所有可用的字体供用户选择。在这里也可以设定粗体、斜体等属性。如图4-24所示。



图4-24 字体对话框



字体对话框中除包含了字体信息外，还包含了颜色等其他信息。下面的代码演示了如何获得并使用这些信息(代码见光盘：\book.ch4.UsingFontDialog.java)。

```
FontDialog fd = new FontDialog(shell);
FontData data = fd.open(); 取得颜色信息
```

```
RGB fontColor = fd.getRGB(); 使用字体对话框返回的信息来创建一个Font实例
```

```
if(data != null) {
```


```
Font font = new Font(Display.getDefault(), data);
```

```
Color color = new Color(Display.getDefault(),fontColor);
//使用字体和颜色信息显示文字
color.dispose();
```

```
font.dispose();
```

4.11 本章小结

本章以代码结合图示的方式，将常用的控件以及对话框的使用作了概略的介绍，对它们所触发的事件也予以了说明。这一章的知识是SWT界面编程中的基础，掌握了基本控件和对话框的使用后，下一章将介绍一类特殊的控件——容器的使用。



看清楚，这是成功的开始!

第5章 容器与布局管理器

拉开崭新的学习帷幕

上一章介绍了各种常用的基本控件，在创建这些基本控件时，都需要提供一个父控件(如Shell、Composite等)来容纳它们，这些能够容纳其他控件的控件被称为容器类控件，它们较少同用户进行直接交互，而主要是用于划分屏幕区域，将其他控件分组排布等目的。在容器类控件中，还有一个重要的概念叫做布局管理器，它专门用于管理容器中的控件的位置、尺寸和排列。本章将着重介绍SWT中常用的几种容器，说明如何设置容器外观、上下文菜单等技术，最后将介绍常用的布局管理器，并讲述如何自定义一个布局管理器。

本章内容包括：

- ★ Composite和Group。
- ★ Shell。
- ★ 容器上下文菜单设置。
- ★ 容器颜色、背景和鼠标指针设置。
- ★ 布局管理概述。

进入第05章



5.1 Composite

Composite容器是SWT控件体系中最基本的容器类型，是所有其他容器类型的父类。

Composite的设计思想基于设计模式中的Composite模式，利用容器来包含其他容器或基本控件，可以将控件组合成树状的结构。一个Composite容器中可以包含任意多的基本控件或子容器控件，父容器像处理基本控件一样对子容器发送各种消息，而子容器在接收到这些消息后，再负责通知包含在其中的控件。这种层状结构有利于设计模块化的程序，能够方便地应对结构的变化，也为系统提供了较好的扩展性，是几乎所有的GUI设计系统都会使用的模式。如果读者有AWT或Swing的开发经验，可以将Composite类比于AWT中的Container和Swing中的JPanel来理解。图5-1显示了这种层状结构。

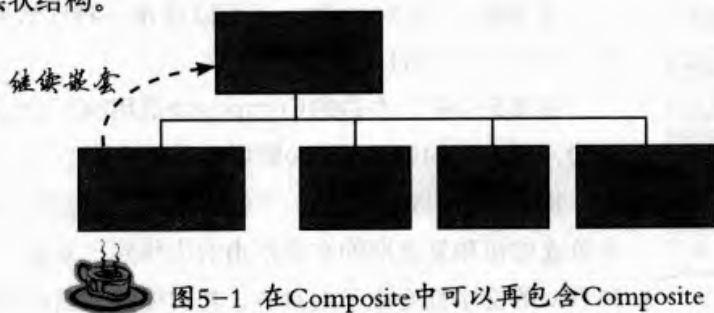


图5-1 在Composite中可以再包含Composite

与基本控件一样，在创建Composite时，也需要一个父控件。下面的代码在一个Shell中创建了一个有边框的Composite(代码见光盘：\book.ch5.CreateComposite.java)，效果如图5-2所示。

```
shell.setSize(100,120);  
  
Composite composite = new Composite(shell,SWT.BORDER);  
composite.setBounds(10,10,90,40);
```

Composite控件
在界面上划分
出了一块区域

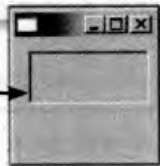
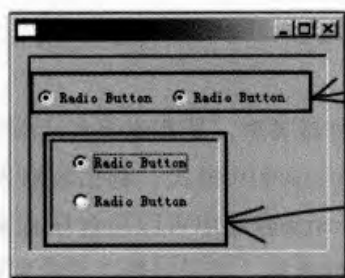


图5-2 有边框的Composite

使用有边框的Composite可以很轻松地将其中的内容同用户界面的其他部分区分开来，而使用没有边框的Composite则有助于将容器中的内容无缝地集成到界面上。在界面设计时，可以灵活使用。

对包含在其中的Radio Button，Composite容器默认提供了排他性，即包含在同一个Composite容器中的Radio Button最多只能有一个处于选中状态。这种默认行为为使用Radio Button提供了方便，但是在使用中需要注意，这种排他性只会应用于直接包含在这个Composite中的Radio Button，而不会应用到子容器中的Radio Button。在创建Composite对象时，通过设定样式NO_RADIO_GROUP可以改变Composite的这一默认行为，允许其中的Radio Button被同时选中，如下面代码所示(代码见光盘：\book.ch5.NoRadioComposite.java)，效果如图5-3所示。



外层的 Composite 使用了 `NO_RADIO_GROUP` 样式，两个 `Radio Button` 可以同时被选中

内层的 Composite 没有使用 `NO_RADIO_GROUP` 样式，因此只有一个 `Radio Button` 可以被选中



图5-3 Radio Button示例

```
Composite child = new Composite(parent, SWT.BORDER | SWT.NO_RADIO_GROUP);
```

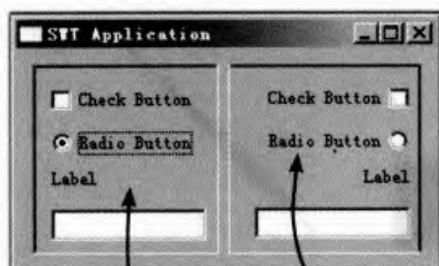


图5-4 RIGHT_TO_LEFT样式

在创建 Composite 时，还可以使用一种比较有趣的样式，即 `RIGHT_TO_LEFT`。

在图5-4中，左边的 Composite 是用 `BORDER` 样式创建的，而右边的 Composite 则增加了 `RIGHT_TO_LEFT` 样式，其余完全相同。对比一下就会发现，右边的 Composite 中单选按钮和复选框的文字都由右边移到了左边，而 Label 和 Text 的文字都变成了右对齐，其实现如下面代码所示。

`RIGHT_TO_LEFT` 的样式是为了满足某些国家和地区右对齐的特殊书写与阅读习惯而设计的。需要注意的是，这种样式会影响到所有包含在容器内的控件。而且如果作为父容器的 Composite 使用了 `RIGHT_TO_LEFT` 样式，子容器即使没有使用该样式，包含在子容器内的控件也一样会按照右对齐的方式显示出来。

```
Composite left = new Composite(parent, SWT.BORDER);
```

```
Composite right = new Composite(parent, SWT.BORDER|SWT.RIGHT_TO_LEFT);
```

`EMBEDDED` 样式的用法与其他不同，它的目的不在于调整 Composite 的显示样式，而更类似于一种标志，使用这种样式的 Composite 容器通常不用来摆放其他 SWT 控件，而是用来显示非 SWT（如 AWT，Swing 等）的 GUI 内容。`EMBEDDED` 样式的 Composite 在 SWT 与其他 GUI 内容中起到了桥梁的作用，如图5-5所示。

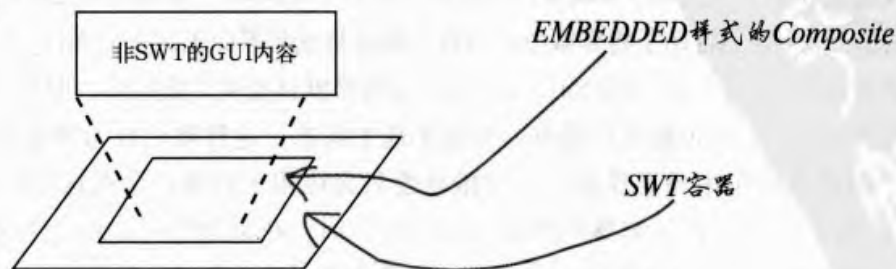


图5-5 EMBEDDED的Composite起了桥梁的作用

将相关的基本控件用Composite容器组合起来构成复合控件，不仅能够使界面更加干净整齐，对于开发人员来说也是一种好的编程习惯，这使得代码更加易于理解和修改，也能够更好地应对需求的变动。

如果有一组相关的控件，它们之间有复杂的相互控制逻辑，而且这组控件在系统中又会被比较频繁地使用到，推荐的解决方案是创建一个Composite的子类，将这些控件添加进去，编写事件处理代码控制它们，并且在其他地方把这个子类当作一个基本控件来使用，这也是实现图形界面代码重用的一个好方法。下面就通过一个实例来看一下如何利用Composite包装GUI代码，以创建自定义的控件。

要创建的控件由一个只读的Text控件和一个Button组成，Text中显示着Button被单击过的次数。



实现代码如下所示(代码见光盘: \book.ch5.MyControl.java)。

```

public class MyControl extends Composite {
    private Button button;
    private Text text;
    private int count = 0;

    public MyControl(Composite parent, int style) {
        super(parent, style);
        setLayout(null);

        text = new Text(this, SWT.READ_ONLY | SWT.BORDER);
        text.setBounds(5, 5, 50, 20);
        text.setText(String.valueOf(count));

        button = new Button(this, SWT.NONE);
        button.setBounds(60, 5, 50, 20);
        button.addSelectionListener(new SelectionAdapter() {

            public void widgetSelected(final SelectionEvent e) {

                text.setText(String.valueOf(++count));

            }
        });
        button.setText("button");
    }
}
    
```

在Composite中创建Text

在Composite中创建Button

当Button被单击时，将计数器增加1，并将Text的文本设成计数器的值

上面的代码创建了一个名为MyControl的控件，使用它的方法和使用基本控件一样，代码如下所示(代码见光盘: \book.ch5.UsingMyControl.java)，效果如图5-6所示。

```

MyControl control1 = new MyControl(shell, SWT.NONE);
control1.setBounds(10, 10, 120, 25);

MyControl control2 = new MyControl(shell, SWT.NONE);
control2.setBounds(10, 40, 120, 25);

MyControl control3 = new MyControl(shell, SWT.NONE);
control3.setBounds(10, 70, 120, 25);
    
```

使用自定义控件
与使用普通控件
的方法没有区别

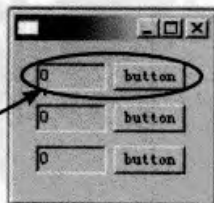


图5-6 使用自定义控件

5.2 Group

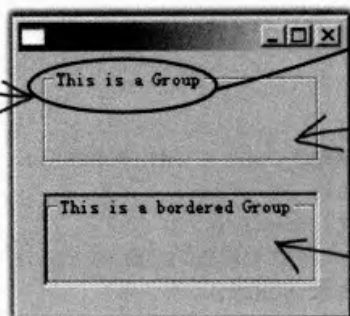
Group容器是Composite容器的子类，直接继承自Composite容器，它为包含在其中的控件提供了默认的边框，并且可以支持在边框左上方显示一行文字标题。

可以用于Composite的样式，如BORDER, NO_RADIO_GROUP和RIGHT_TO_LEFT等都可以直接应用于Group容器。如图5-7所示，在Group容器上使用BORDER样式会使Group除自带的边框外还在外层显示出下凹的边框。下面的代码演示了如何创建group并设定标题(代码见光盘：\book.ch5.CreatingGroup.java)。

```
final Group group = new Group(shell, SWT.NONE);
group.setText("This is a Group");
group.setBounds(15, 10, 173, 60);

final Group borderedGroup = new Group(shell, SWT.BORDER);
borderedGroup.setText("This is a bordered Group");
borderedGroup.setBounds(15, 89, 173, 60);
```

setText设定
Group的标题
文字



创建一个默认样式
的Group(浅边框)

BORDER样式的Group不常用



图5-7 创建Group容器

在设计用户界面时，对于需要唯一选择的情况，通常用一组Radio Button来表示所有可选项，并将这些Radio Button摆放在同一个Group中，同时在Group的标题中对这个问题简要地进行介绍。如图5-8所示是来自Windows XP的一个例子。

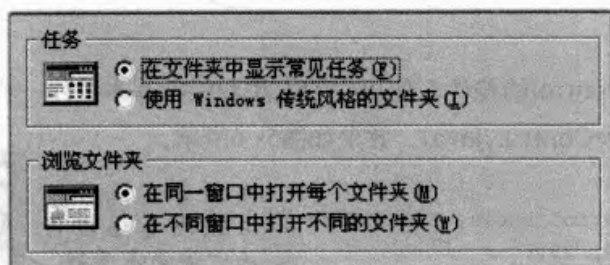


图5-8 使用Group



5.3 Shell

无论是Composite还是Group, 都是创建一个父容器里面的, 那么必然会有一个顶层容器可以不需要父容器而存在, 代表窗口的Shell, 就是这个最顶层的容器。Shell是Composite容器的子类, 但它只能作为顶层的父容器出现, 而无法被嵌套在其他容器内。第3章已经介绍过一些Shell的知识, 这一节将从容器的角度再对它进行探讨。

创建Shell时, 有着比创建Composite更多的样式可以选择, 但在使用这些花样繁多的样式时, 有一点一定要注意, 由于Shell是直接与管理操作系统的窗口挂钩的, 这些样式的实现, 都依赖于操作系统是否提供了对应的功能。如果使用了操作系统不支持的样式, 就不会产生任何效果。即使是同样的样式, 由于不同操作系统的图形平台具体实现不同, 也可能产生不同的效果。因此如果使用SWT开发跨平台的程序, 在设置Shell容器的样式时一定要慎之又慎。

下面首先来看在上面两个例子中用到的SHELL_TRIM样式。SHELL_TRIM是一种复合样式, 它由CLOSE, TITLE, MIN, MAX和RESIZE几种基本样式组成。SHELL_TRIM是默认和普通窗口样式。CLOSE, MIN和MAX样式分别控制着是否显示窗口上的关闭、最小化和最大化按钮。TITLE控制着是否显示标题栏, 而RESIZE则控制着窗口的大小是否能够被改变。需要注意的是, 由于最小化、最大化和关闭按钮都显示在标题栏上, 因此如果使用了CLOSE, MIN或MAX样式, 无论是否使用TITLE样式, 标题栏都会出现。图5-9由左到右分别显示了使用SHELL_TRIM, CLOSE和RESIZE样式创建的窗口。读者可以自行试验其余的样式组合。



SHELL_TRIM样式
窗口有关闭、最小化和最大化按钮, 可以调整尺寸



CLOSE样式
窗口只有关闭按钮, 不能调整尺寸



RESIZE样式
窗口没有标题栏, 可以调整尺寸



图5-9 Shell的样式

另外一种比较常用的复合样式是DIALOG_TRIM, 从名字就可以看出来, 这种样式包含了对话框窗口常用的基本样式。它是由TITLE, CLOSE和BORDER样式组成的。TITLE和CLOSE样式上面已经提到过, 而BORDER样式则为窗口提供了一个明显的边界。

图5-10显示的窗口与通常所见到的窗口不同, 它的标题栏比较窄, 字体也要小几号, 其实现如图下方代码所示。这是利用TOOL样式创建的窗口, 如果一个窗口同时使用了TOOL样式与TITLE样式, 它的标题栏就会以小于正常尺寸的样子显示出来。在程序中, 这种窗口通常作为浮动工具栏出现, 这也正是它被称为TOOL的原因。



字体和按钮尺寸都比正常情况要小。这个尺寸是系统默认，无法通过程序调节的

```
Shell shell = new Shell(display, SWT.TOOL, SWT.CLOSE);
```



图5-10 创建TOOL样式的Shell



无论是MAX, MIN, CLOSE还是TITLE, BORDER, RESIZE, 它们的作用都是在窗口上添加装饰组件, 如关闭按钮、边框等, 所以这一类样式有个共同的名字, 称为装饰类样式。但是NO_TRIM样式的作用却恰恰相反, 使用了NO_TRIM的窗口, 会被去掉所有的装饰组件, 如果NO_TRIM和其他装饰类的样式共同应用于一个Shell, 所有装饰类的样式都会失效, Shell会显示成一个没有边框, 没有标题栏的窗口, 代码如下所示。

```
Shell shell = new Shell(display, SWT.SHELL_TRIM|SWT.NO_TRIM);
```

某些媒体播放器窗口在播放视频的时候, 无论打开多少窗口, 它都是浮在最上层的。SWT提供了ON_TOP样式以实现这一功能, ON_TOP样式可以附加到任意样式的窗口上, 它的唯一作用就是使窗口永远处于用户桌面的最上层, 代码如下所示。

```
Shell shell = new Shell(display, SWT.SHELL_TRIM|SWT.ON_TOP);
```

Shell的模态 (Modality) 样式是一类比较特殊的样式。程序中常常有这样的情况, 当打开某个对话框后, 再试图操作对话框的父窗口时, 父窗口却不做任何响应, 这通常是因为对话框窗口阻截了发送到其父窗口的操作消息造成的, 可以通过设置Shell容器的模态样式来决定其是否阻截这些操作消息。可用的模态样式有4种, 分别为MODELESS, PRIMARY_MODAL, APPLICATION_MODAL和SYSTEM_MODAL, 对同一个Shell容器, 只能选择使用一种模态样式。MODELESS样式使子窗口不对任何消息进行阻截, 这也是Shell的默认模态样式; PRIMARY_MODAL样式使得子窗口阻截所有发送到其父窗口的操作消息, 如果不关掉这个窗口, 就无法对父窗口进行操作, 这种模态常见于对话框窗口; APPLICATION_MODAL样式使子窗口阻截所有发送到与它共享一个Display实例的窗口的操作消息, 如果一个应用程序只拥有一个Display实例 (实际应用中这是最常见的), 那么不关掉这个窗口, 用户便无法使用这个应用程序的其他窗口, 这正是它被命名为APPLICATION_MODAL样式的原因; 最后一种是SYSTEM_MODAL, 它允许窗口阻截操作系统中所有其他窗口的消息, 使用了这种样式, 意味着如果不关掉这个窗口, 用户将无法操作桌面上所有其他应用程序, 这是一种在实际应用中极少需要用到的样式。由于这种样式会导致其他桌面程序被冻结, 很多图形平台已经不再支持SYSTEM_MODAL样式的使用, 这时使用了SYSTEM_MODAL样式的程序会被SWT自动转为使用APPLICATION_MODAL样式。下面的程序演示了PRIMARY_MODAL和APPLICATION_MODAL的使用(代码见光盘: \book.ch5.ShellModality.java)。

```
shell shell1 = new Shell(display);
shell1.setText("Shell 1");
shell1.setSize(206, 149);
```

```
Shell shell2 = new Shell(display);
shell2.setSize(206, 150);
shell2.setText("Shell 2");
```

```
shell1.open();
shell2.open();
```

同一个程序中的两个窗口

```
Shell primaryShell = new Shell(shell1, SWT.DIALOG_TRIM
    | SWT.PRIMARY_MODAL);
primaryShell.setText("This window will block shell 1 only");
primaryShell.setSize(300, 80);
primaryShell.open();
```

使用PRIMARY_MODAL的窗口只会阻挡其父窗口(shell1)

```
Shell applicationShell = new Shell(shell1, SWT.DIALOG_TRIM
    | SWT.APPLICATION_MODAL);
applicationShell.setText("This window will block both");
applicationShell.setSize(300, 80);
applicationShell.open();
```

使用APPLICATION_MODAL的窗口会阻挡程序中的所有窗口(shell1和shell2)

Shell容器支持5种事件, 分别为Activate, Close, Deactivate, Deiconify和Iconify, 这些事件都可以通过向Shell添加实现了ShellListener接口的监听器而捕捉到。

在一个Shell窗口成为活动窗口后, 会触发Activate事件。Active事件可以在窗口被创建出来时触发, 也可以在用户的输入焦点落在Shell窗口时触发。

当一个Shell窗口被关闭后, 会触发Close事件。

当Shell从活动窗口变成非活动窗口时, 会触发Deactivate事件。

当Shell从最小化状态恢复时, 会触发Deiconify事件。

当Shell被最小化时, 会触发Iconify事件。

下面的代码演示了如何使用ShellListener来监听Shell事件(代码见光盘: \book.ch5. MonitorShell.java)。

.....

```
final Text text = new Text(shell, SWT.READ_ONLY | SWT.MULTI
    | SWT.BORDER | SWT.V_SCROLL);
text.setBounds(10, 10, 380, 150);
shell.addShellListener(new ShellListener() {
    public void shellActivated(ShellEvent e) {
```


当事件发生时，打印一行提示文字到text控件中

```

        text.append("Shell has been activated\n");
    }

    public void shellClosed(ShellEvent e) {
        System.out.println("Shell has been closed");
    }

    public void shellDeactivated(ShellEvent e) {
        text.append("Shell has been deactivated\n");
    }

    public void shellDeiconified(ShellEvent e) {
        text.append("Shell has been deiconified\n");
    }

    public void shellIconified(ShellEvent e) {
        text.append("Shell has been iconified\n");
    }
}

```

窗口关闭事件发生后，就不能用text控件显示提示了，因此将提示打印到控制台

运行这段代码，然后可以尝试切换窗口状态(最大化，最小化，激活等)，效果如图5-11所示。

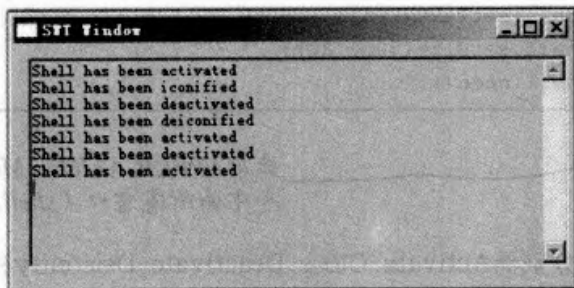


图5-11 监听Shell的事件



5.4 容器上下文菜单设置

容器的上下文菜单(Context Menu)，是指在容器上单击鼠标右键时弹出的菜单，这个菜单的内容通常是与当前位置最相关的操作选项。单击不同的容器，弹出的菜单可能有所不同，因此被称为上下文菜单。

图5-12是一个上下文菜单的例子。右键单击左边的容器和右边的容器，出现的是不同的上下文菜单。

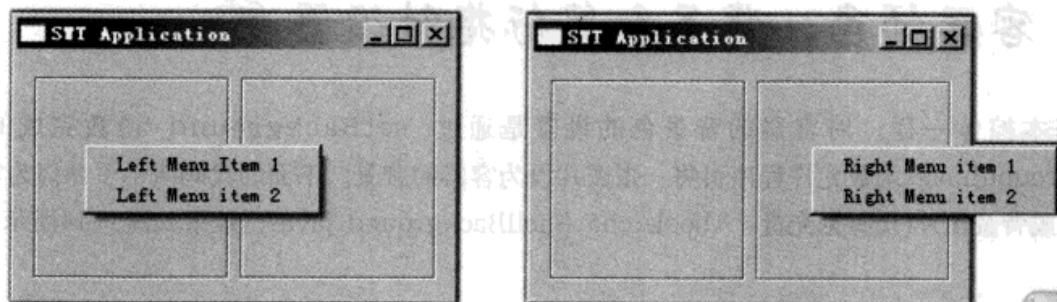


图5-12 上下文菜单示例

创建上下文菜单时，首先要以Shell为父容器创建一个样式为POP_UP的Menu，然后调用Control类的setMenu方法将创建的Menu与这个容器相关联。下面的代码演示了如何为一个Shell创建弹出式上下文菜单(代码见光盘：\book.ch5.CreatingShellContextMenu.java)，效果如图5-13所示。

.....

```
Menu menu = new Menu(shell, SWT.POP_UP);
```

```
container.setMenu(menu);
```

```
MenuItem menuItem1 = new MenuItem(menu, SWT.NONE);
menuItem1.setText("Menu item1");
menuItem1.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
        MessageBox mb = new
        MessageBox(shell, SWT.ICON_INFORMATION|SWT.OK);
        mb.setText("Notification");
        mb.setMessage("Item 1 has been selected!");
        mb.open();
    }
});
```

```
MenuItem menuItem2 = new MenuItem(menu, SWT.NONE);
menuItem2.setText("Menu item2");
.....
```

调用setMenu方法将菜单与容器关联起来

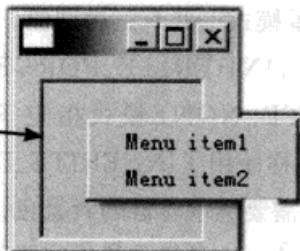


图5-13 为Shell添加上下文菜单

在创建Menu的时候，也可以直接调用Menu(Control control)的构造函数。这个构造函数会自己寻找Control实例最上层的Shell容器，并以POP_UP为默认style参数调用Menu(Shell shell, int style)的构造函数，因此这里也可以写成Menu menu = new Menu(container)，效果完全一样

关于如何创建Menu的内容，可以参见第4.7节



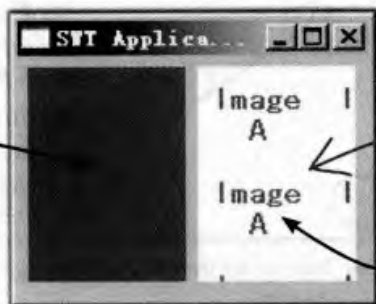
5.5 容器颜色、背景和鼠标指针设置

和基本控件一样，对容器的背景色的设置是通过 `setBackground` 函数完成的，而 `setBackgroundImage` 函数允许程序员将一张图片设为容器的背景。下面的代码演示了如何为容器设定背景色或背景图片(代码见光盘：\book.ch5.ShellBackground.java)，效果如图5-14所示。

```
final Composite composite = new Composite(this, SWT.NONE);
composite.setBackground(Display.getCurrent().getSystemColor(SWT.COLOR_RED));
composite.setBounds(5, 5, 80, 110);
```

```
final Composite composite_1 = new Composite(this, SWT.NONE);
composite_1.setBackgroundImage(SWTResourceManager.getImage(ShellBackground.
class, "image1.bmp"));
composite_1.setBounds(91, 5, 80, 110);
```

将背景色设置
为红色



如果背景图片的尺寸小于
容器尺寸，容器会重复地
将图片平铺在其中

将一张图片设置
为容器背景



图5-14 容器的背景色和背景图片

※ 注意: ※

一般来说为容器设置前景色是没用的，但是如果需要在容器中直接绘图，就会用到前景色了。可以参见第8.2.3节中在Shell内绘图的例子。



除了背景色和背景图片外，容器还有一个涉及背景色的属性称为背景模式 (Background Mode)。这个参数并不影响当前容器的背景显示，而是对当前容器的子控件发生作用，它决定子控件是否要继承容器的背景色设定。背景模式一共有三种取值，分别为 `SWT.INHERITE_NONE`，`SWT.INHERITE_DEFAULT` 和 `SWT.INHERITE_FORCE`。可以用 `setBackgroundMode` 来改变容器的背景模式设置。 `INHERITE_NONE` 模式指定子控件不继承容器的背景色设定。 `INHERITE_DEFAULT` 模式指定那些同样使用了 `INHERITE_DEFAULT` 模式的子控件才会继承容器的背景色设定。举例来说，Label 通常需要继承容器的背景色设定（否则会显示得像一块难看的补丁），而 Table 通常会选择自己设定背景色。这种模式使得容器将继承的决定权交给了子控件，相对另外两种模式来说更灵活也更常用。 `INHERITE_FORCE` 模式指定子控件要继承容器的背景色设定。

下面的示例演示了不同的模式的表现。当容器使用INHERITE_NONE背景模式时，Label和Text都保持了它们自己的背景色设定；当INHERITE_DEFAULT模式被使用时，Label继承了父容器的背景色而Text没有；当使用了INHERITE_FORCE模式时，Text和Label的背景色都被强制设定成和父容器的背景色一致。如图5-15所示(代码见光盘：\book.ch5.ShellBackgroundMode.java)。

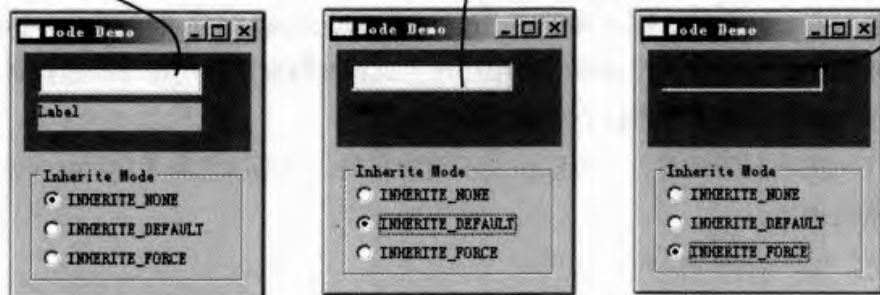


图5-15 容器的不同背景模式

除了设置容器的背景和图片，还有另外一种方法可以吸引用户的注意力，那就是为容器设置鼠标指针，这可以使鼠标移动到容器上的时候，指针变成指定的样式。除了可以使用常见的手形，十字形等指针外，也可以使用自定义的图片作为鼠标指针。为容器设定特定的鼠标指针，可以用setCursor函数完成。下面一段代码将Composite容器的鼠标指针设成了手形。

```
Cursor handCursor = new Cursor(Display.getDefault(), SWT.CURSOR_CROSS);
composite.setCursor(handCursor);
.....
handCursor.dispose();
```

使用后释放cursor资源

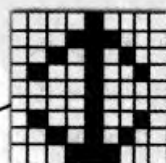
使用CURSOR_CROSS, CURSOR_HELP等代替CURSOR_CROSS就可以创建其他样式的指针

另外，也可以使用Display.getSystemCursor(SWT.CURSOR_CROSS)来得到一个Cursor实例，这样得到的Cursor不需要释放。

创建一个使用图片自定义的鼠标指针稍微有所不同，需要首先将图片读进来得到一个ImageData实例，然后使用这个ImageData实例创建一个Cursor实例，再将它与容器相关联。下面代码使用一个GIF文件(cursor.gif)为Composite容器创建了一个图片指针(代码见光盘：\book.ch5.CompositeCursor.java)。

```
ImageData cursorData = new ImageData(Demo.class.getResourceAsStream("cursor.gif"));
Cursor cursor2 = new Cursor(display, cursorData, 0, 0);
//后面的两个数字表明了以图片上的哪一点来作为鼠标指针的箭头所在
composite.setCursor(cursor2);
//正常运行程序，最后释放Cursor资源
cursor2.dispose();
```

这一点相当于鼠标箭头的尖端，各种鼠标事件的坐标都是以它的位置决定的



作为指针的图片

5.6 布局管理概述

对一个容器而言，布局（Layout）就是指包含在其中的子控件的位置和尺寸，布局决定着容器中每一个控件应该放在什么位置，而布局管理（Layout Management）则负责在容器的尺寸发生变化时，按照布局重新设置其中的子控件的位置和尺寸。只有按照容器的用途合理选择布局，才能使应用程序的界面在不同的窗口尺寸下都有干净整洁的外表。

为容器安装不同的布局管理器，就可以实现不同的布局。下面的代码为Composite容器安装了一个GridLayout布局管理器。

```
composite.setLayout(new GridLayout());
```

安装了布局管理器后，为了通知布局管理器开始安排子控件，需要在容器上调用layout方法。这个方法除了通知布局管理器开始安排所有子控件的位置外，还会自动寻找当前容器中所有子容器并调用它们的layout方法。因此，只需要在最高层的容器(Shell)上调用一次layout就可以了。当改变了容器的布局管理器时，也需要重新调用layout方法。

所有的布局管理器都必须继承类org.eclipse.swt.widgets.Layout。SWT自带的布局管理器有FillLayout, GridLayout, StackLayout等，本节将对这些常见布局管理器以及如何选择布局或自定义布局进行介绍。

在介绍布局之前，还要提到一个概念——控件的偏好尺寸(Preferred Size)。偏好尺寸是控件按照自己的属性，如其中文字的长度等自行计算出的最佳显示尺寸，在这个尺寸下控件的显示效果最好。调用Control.pack()方法可以将控件恢复到它的偏好尺寸，许多布局类在默认情况下都会基于控件的偏好尺寸进行排列。

5.6.1 AbsoluteLayout布局 (No Layout)

如果没有为容器指定布局管理器，容器就会使用默认的AbsoluteLayout。在这种布局下面控件的位置和尺寸都是固定的，不会随父容器的尺寸变化而变。代码如下所示。

```
composite.setLayout(null);
```

使用AbsoluteLayout时，在创建一个子控件后，必须调用setBounds指定一个矩形区域或setLocation和setSize指定矩形的左上顶点及宽度和高度确定用于绘制这个子控件的位置，否则这个控件将无法显示在屏幕上。之前的示例代码中，都使用了AbsoluteLayout。

顶点坐标的确定如图5-16所示，坐标的单位是像素。在坐标系中，通常使用一组的4个数字来确定一个矩形，如(x, y, weight, height)。其中的(x, y)是控件左上角的坐标，而weight和height则是控件的宽度和高度。需要注意的是，Location的坐标可以为负值，只要矩形有一部分落在父容器的显示区域内(第一象限)，子控件还是可以显示出一部分，但是如果将weight或height设置成了负值，那么子控件是显示不出来的。可以用Control.setBounds方法设定这些值。下面一段示例代码将一个Button控件左上角放置在点(10, 10)处，并设定其宽、高分别为100和25像素。

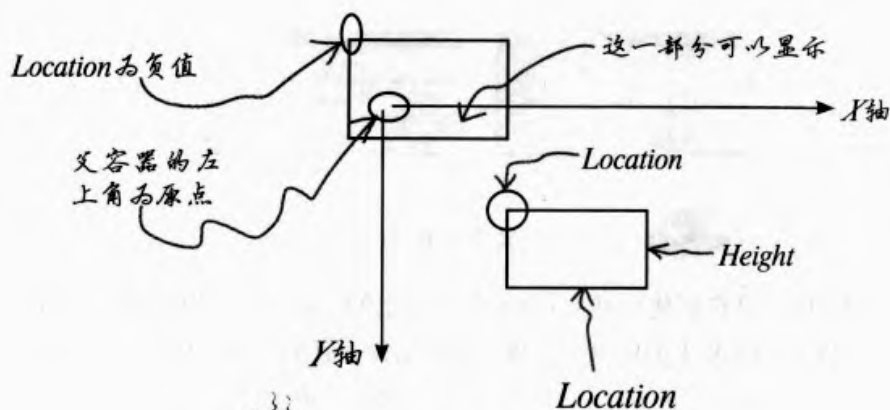


图 5-16 容器内的坐标系

※ 注意: ※

使用除`AbsoluteLayout`之外的布局管理器时，不需要为控件设定尺寸和位置。即使设定了也是无效的，管理器会自行完成这些工作并覆盖用户的设置。

5.6.2 FillLayout 布局

`FillLayout`是一种最简单的`Layout`布局，它将所有的子控件按照创建的顺序自左到右横向排布（排在同一行中）或自上到下纵向排布（排在同一列中），并给它们平均分配父容器的可视区域。所有的控件加起来会充满整个父容器的可视区域，这就是这种布局被称为`FillLayout`的原因。`FillLayout`保证容器中所有子控件都会显示出来，但使用`FillLayout`时，开发者无法手动指定子控件的大小。而且无论窗口缩放到什么程度，所有的控件都会保持在同一行或同一列中。下面的代码对一个`Shell`容器使用`FillLayout`并在其中加入了三个按钮（代码见光盘：`\book.ch5.UsingFillLayout.java`），效果如图5-17所示。

```
Display display = Display.getDefault();
Shell shell = new Shell(display, SWT.SHELL_TRIM);
shell.setText("FillLayout Demo");
shell.setSize(300, 150);
```

设定使用`FillLayout`

```
FillLayout layout = new FillLayout();
shell.setLayout(layout);
```

```
Button button = new Button(shell, SWT.NONE);
button.setText("button1");
```

```
Button button_1 = new Button(shell, SWT.NONE);
button_1.setText("button2");
```

```
Button button_2 = new Button(shell, SWT.NONE);
button_2.setText("button3");
```

直接创建控件，不需要指定位置和尺寸，这些工作交给布局管理器来完成

```
shell.open();
shell.layout();
```

调用`layout`方法，通知容器开始布局

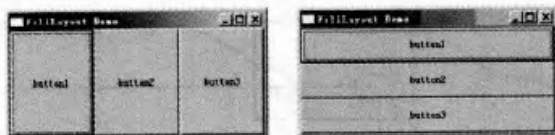


图5-17 横向和纵向的FillLayout

FillLayout排布的方向，是在创建FillLayout对象时指定的。如果创建时没有指定方向，默认的方向是横向，图5-17中右图显示了将上面的例子改成使用纵向排布时的效果，代码如下所示。

```
FillLayout horizontalLayout = new FillLayout(); //默认，横向
FillLayout horizontalLayout1 = new FillLayout(SWT.HORIZONTAL); //横向
FillLayout verticalLayout = new FillLayout(SWT.VERTICAL); //纵向
```

除方向外，FillLayout还有几个比较重要的属性，横向页边空白(marginWeight)、纵向页边空白(marginHeight)和间距(spacing)。如图5-18所示的示意图对这几个属性作了说明。在其他Layout中，也同样可以设定页边空白和控件间距。

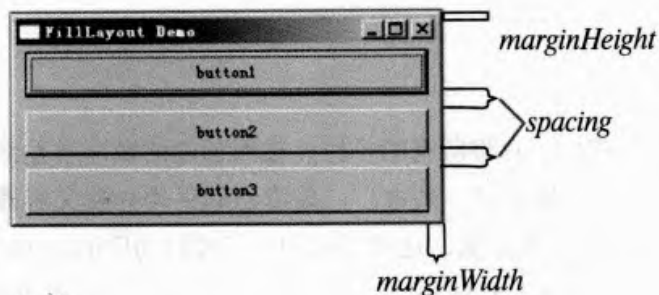


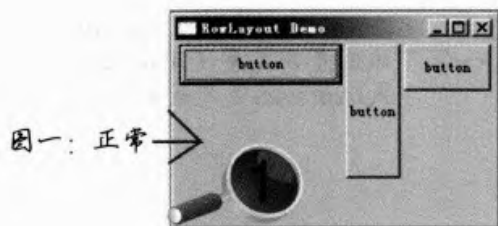
图5-18 FillLayout的间距属性

下面这段代码将FillLayout设置成横向页边空白为10，纵向页边空白为5，间距为8。

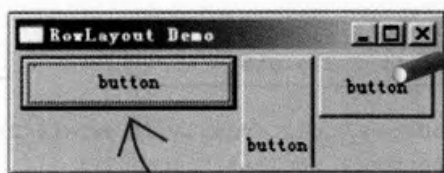
```
FillLayout layout = ;
layout.spacing = 8;
layout.marginWidth = 10;
layout.marginHeight = 5;
```

5.6.3 RowLayout布局

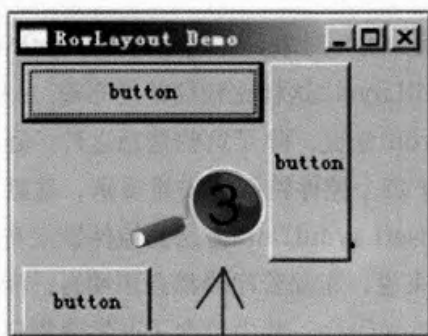
与FillLayout相似，RowLayout也会尝试将子控件安排在同一行或同一列中，但是RowLayout允许开发者为每一个子控件指定宽度和高度，如果没有指定，RowLayout会使用控件的偏好尺寸。当子控件无法全部放置在一行(列)中时(在横向排布时，子控件的宽度之和超过了容器的宽度；或者在纵向排布时，子控件的高度之和超过容器的高度)，RowLayout会尝试换行，每一行的高度是这一行中所有子控件的高度最大值。RowLayout并不保证所有的子控件都能显示出来。下面的代码演示了一个使用横向RowLayout的窗口中的控件在窗口尺寸变化时位置的变化(代码见光盘：\book.ch5.UsingRowLayout.java)，效果如图5-19所示。



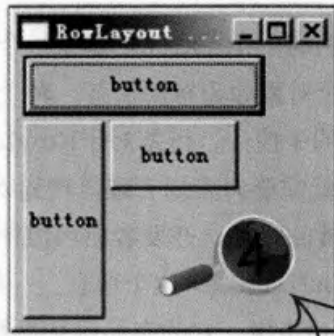
图一：正常



图二：横向排列的RowLayout对容器的高度变化没有反应



图三：缩小窗口宽度，一行的宽度容不下所有控件，第三个button换到了第二行，第一行的高度是由最高的button确定的



图四：宽度继续缩小，第二个button也换到了第二行，第一行的高度变成由第一个button决定



图五：三个button各占一行，第一个button宽度超过窗口宽度，有一部分显示不出来



图5-19 RowLayout示例

```
Display display = Display.getDefault();
Shell shell = new Shell(display, SWT.SHELL_TRIM);
shell.setText("RowLayout Demo");
shell.setSize(261, 174);
```

```
shell.setLayout(new RowLayout());
```

可以用SWT.HORIZONTAL或SWT.VERTICAL指定方向。不指定方向的话，默认为横向

```
Button button = new Button(shell, SWT.NONE);
final RowData rowData = new RowData();
rowData.height = 32;
rowData.width = 128;
button.setLayoutData(rowData);
button.setText("button");
```

```
Button button_1 = new Button(shell, SWT.NONE);
```

```
final RowData rowData_1 = new RowData();
rowData_1.height = 107;
rowData_1.width = 43;
button_1.setLayoutData(rowData_1);
```

为每一个控件创建一个RowData对象，其中包含了指示布局管理器应当如何排布这个控件的信息

```
button_1.setText("button");
Button button_2 = new Button(shell, SWT.NONE);
final RowData rowData_2 = new RowData();
```




```
rowData_2.height = 37;
rowData_2.width = 68;
```

在RowData中设定button的高度和宽度(用setSize或setBounds是无效的)

```
button_2.setLayoutData(rowData_2);
button_2.setText("button");
shell.open();
shell.layout();
```

在上面的代码中, 对应每一个button创建了一个RowData对象, 并调用setLayoutData方法将它关联到了button上面, 这个对象称为布局信息。对于FillLayout这样比较简单的布局, 只要知道了容器的大小就可以安排其中的子控件; 但是对于RowLayout来说, 除了这些信息之外, 还需要为每个控件指定其大小, 这时, 就需要向布局管理器传递对于每个控件特殊的安排要求, 这就是布局信息(LayoutData)。在创建控件的时候, 开发者使用控件的setLayoutData方法为控件设定布局信息, 而布局管理器则使用getLayoutData取得这个信息。一般来说, 布局管理器都会声明自己能够识别的布局信息类, 对应RowLayout来说, 它的布局信息类是RowData, 其中包含了开发者指定的子控件宽度和高度。

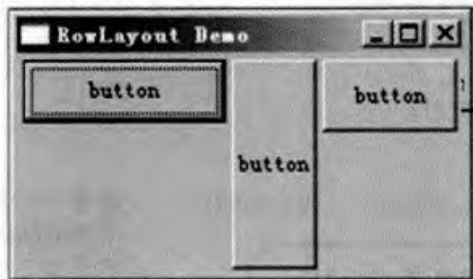
上面看到的是RowLayout的默认行为, 通过改变它的属性的设置, 可以使RowLayout表现得更加符合特定程序的需求。表5-1列出了RowLayout中可以调节的属性。

表 5-1 RowLayout中的属性

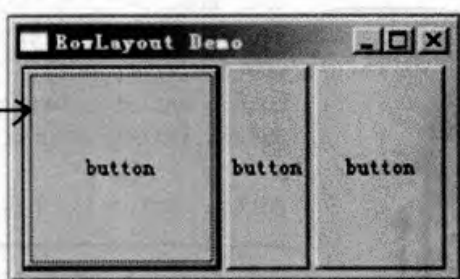
属性名	取值	默认值
fill	true/false	false
justify	true/false	false
pack	true/false	true
wrap	true/false	true

以下的说明均假设RowLayout被设置为横向排列模式。fill属性的设置决定着是否将每一行中所有控件的高度设置成一样(也就是设置成该行中最高的子控件的高度)。图5-20演示了将fill属性设置成true和false时布局的不同表现。

fill = false (默认)



fill = true

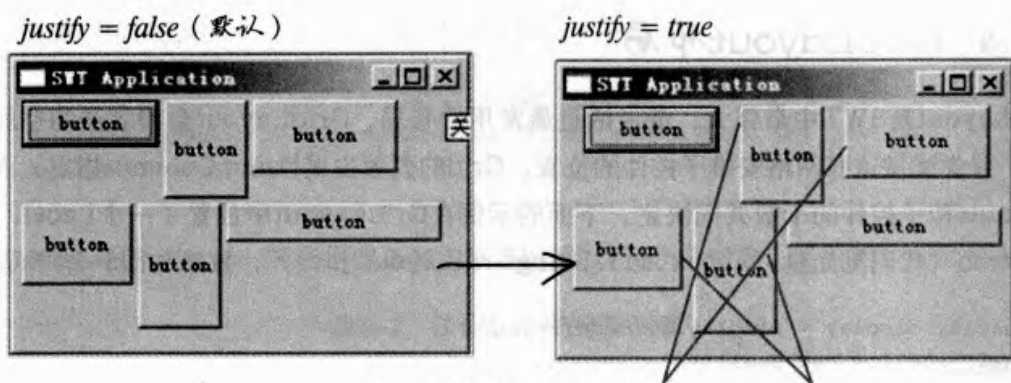


fill属性为true时, 所有的控件被拉伸到和最高的控件一样的高度



图5-20 RowLayout的fill属性

justify属性决定着是否要将每一行中所有控件的间距设置为相等。需要注意的是justify只会保证同一行中各控件的间距相等, 而并不在乎不同行中控件的间距是否一致, 如图5-21所示。



每行中的按钮间距都相等
但不同行的按钮间距相互独立



图5-21 RowLayout的justify属性

wrap属性控制着RowLayout是否会对超出一行宽度的控件自动换行，如图5-22所示。

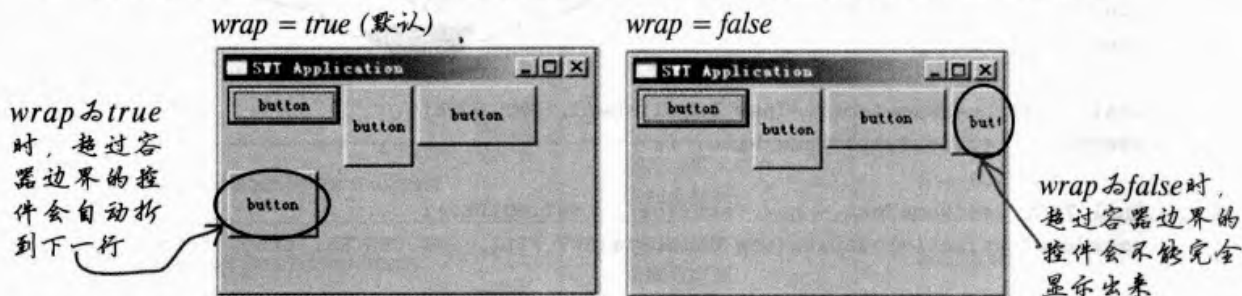


图5-22 RowLayout的wrap属性

pack属性决定了RowLayout是否会使用开发者设置的宽度和高度设置控件的尺寸。如果pack属性被设成false，RowLayout不会为每一个子控件使用预设的宽度和高度，而会将同一行中的控件设置成同样的宽度和高度，其取值分别是该行中子控件的宽度和高度的最大值，如图5-23所示。

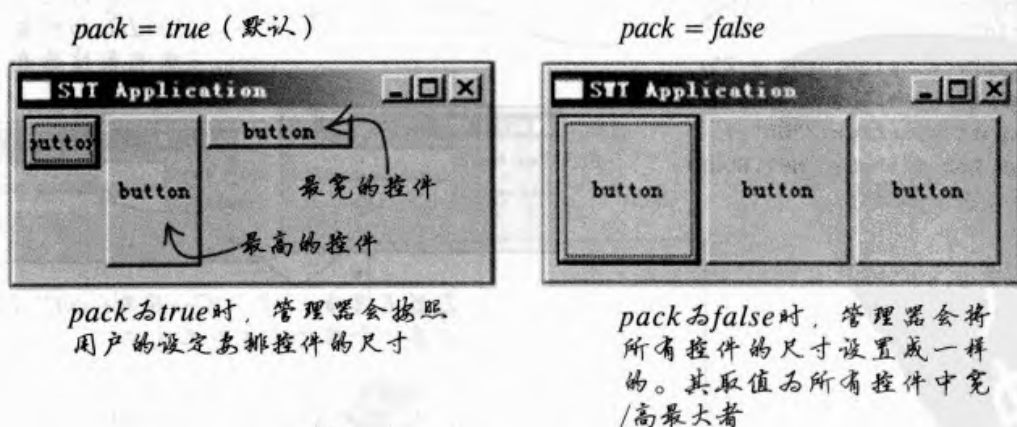


图5-23 RowLayout的pack属性

5.6.4 GridLayout布局

GridLayout是SWT中最强大、最灵活也最常用的布局。GridLayout会将容器的空间划分为网格(Grid),并尝试按这些网格安排子控件的位置,Grid的列数由属性numColumns指定,而行数则由numColumns和子控件的个数共同决定。下面的示例在GridLayout中放置了一个Label,一个Text和一个Button(代码见光盘:\book.ch5.UsingGridLayout.java),效果如图5-24所示。

```

Display display = Display.getDefault();
Shell shell = new Shell();
final GridLayout gridLayout = new GridLayout();

gridLayout.numColumns = 2;

shell.setLayout(gridLayout);
shell.setText("Hello SWT");
shell.setSize(200, 200);
shell.open();

final Label userNameLabel = new Label(shell, SWT.NONE);
userNameLabel.setText("User Name:");

final Text userNameText = new Text(shell, SWT.BORDER);
userNameText.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true, false, 1, 1));

final Label addressLabel = new Label(shell, SWT.NONE);
addressLabel.setText("Address:");

final Text addressText = new Text(shell, SWT.BORDER);
addressText.setLayoutData(new GridData(SWT.FILL, SWT.CENTER,
true, false));

final Button okButton = new Button(shell, SWT.NONE);
final GridData gridData = new GridData(SWT.CENTER, SWT.CENTER,
false, false, 2, 1);
gridData.widthHint = 80;
gridData.heightHint = 25;
okButton.setLayoutData(gridData);
okButton.setText("OK");
new Label(shell, SWT.NONE);
shell.layout();

```

.....



容器长度变化时,可以控制GridLayout使某一个控件长度随之变化



图5-24 使用GridLayout

GridLayout以创建的先后顺序排列控件,并按照由左到右,由上到下的顺序将控件安排在网格中。行高和列宽均由这一行/列中最高/宽的控件决定。由图5-24中可以看到,虽然两个Label的长度

不同，但它们后面的文本框是对齐显示的，其实现代码如下所示。

```
final Text userNameText = new Text(shell, SWT.BORDER);
    userNameText.setLayoutData(new GridData(SWT.FILL, SWT.CENTER,
true, false, 1, 1));
```

GridData就是GridLayout的布局信息类，创建时可以指定6个参数，其内容和含义如表5-2所示。

表5-2 GridData的参数列表

参数	作用	取值
int horizontalAlignment	水平方向如何对齐	SWT.BEGINNING SWT.CENTER SWT.END SWT.FILL
int verticalAlignment	竖直方向如何对齐	SWT.TOP SWT.CENTER SWT.BOTTOM SWT.FILL
Boolean grabExcessHorizontalSpace	是否占用水平方向剩余空间	true/false
Boolean grabExcessVerticalSpace	是否占用竖直方向剩余空间	true/false
int horizontalSpan	水平跨列数	不小于1的整数
int verticalSpan	竖直跨行数	不小于1的整数

前面两行alignment的参数指定了控件在网格中的位置，每个网格的宽/高是由列/行中尺寸最大的控件决定的，那么必然会出现某个控件所在的网格尺寸大于它本身的情况。在这种情况下，控件是靠左上显示，靠右下显示还是居中将由alignment的两个参数控制。它们各自有4个取值，如BEGINNING, TOP等值从名字就可以看出其意义，如图5-25所示。这里只说一下SWT.FILL取值的含义，如果在水平方向上取值为FILL，GridLayout会将这个控件的宽度设为与网格等宽，也就是“充满”网格；竖直方向同理。

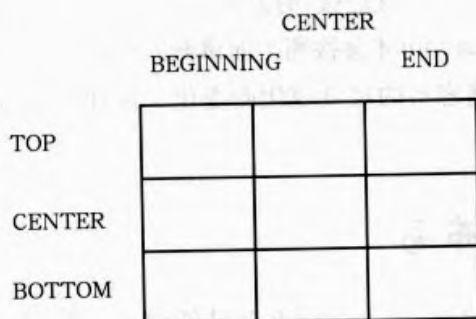


图5-25 每一个网格被分为9部分

由于GridLayout的网格尺寸是由行/列最大尺寸的控件决定的,所以如果所有最大控件的尺寸加起来都不如容器大,就会余留下一些空间,这被称为剩余空间(Excess Space)。控件铺不满容器,会影响界面美观,如果将某一个控件的`gridExcessHorizontalSpace`值指定为`true`,这个控件所在网格的宽度就会扩大到在水平方向上充满整个容器。如果有多个不同列的控件被同时指定为占用水平剩余空间,它们所在的网格将平分剩余空间;竖直方向同理。

默认时,每个控件占用一个网格的空间。在大多数情况下,这能够使控件的排列看起来比较整齐,但是有些时候可能希望为某些控件分配多个网格的空间。这时候可以使用两个`span`属性来指定控件所占用的行/列数。如下面一段代码中,为Button分配了水平方向的两个网格空间并将它的`horizontalAlignment`设为`CENTER`,这样Button就可以处于窗口居中的位置。效果如图5-26所示。

```
final Button okButton = new Button(shell, SWT.NONE);
final GridData gridData = new GridData(SWT.CENTER, SWT.CENTER, false, false,
    2, 1);
gridData.widthHint = 80;
gridData.heightHint = 25;
okButton.setLayoutData(gridData);
```

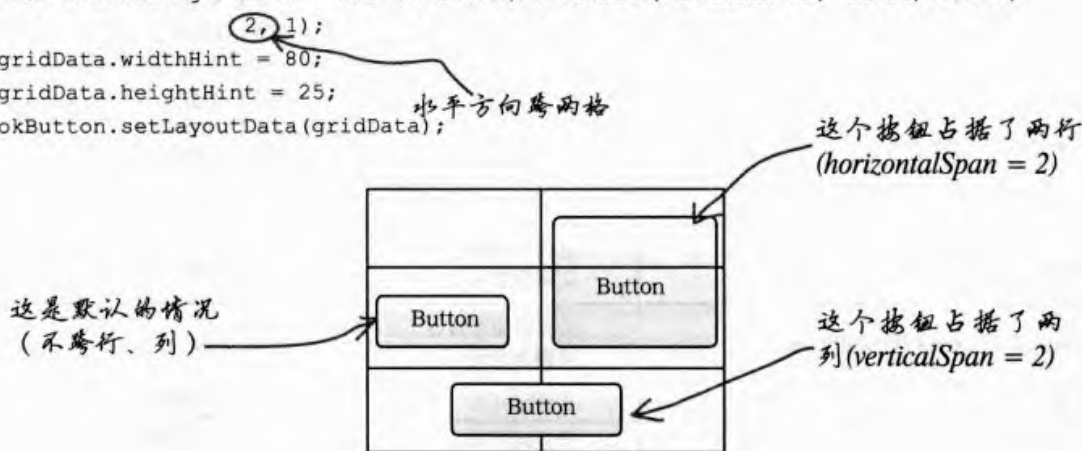


图5-26 GridLayout中跨行/列显示

在上面的代码中,对Button设置了`widthHint`和`heightHint`两个参数。这两个参数允许用户自行定义控件的尺寸。在GridLayout中,不能通过`setSize`或`setBounds`来自定义控件尺寸,GridLayout会按照控件的偏好尺寸(Preferred Size)自动设置它们的大小。如果用户一定要自行指定控件尺寸,就要使用这两个参数来设定。

最后要提到GridData的一个比较危险的属性`exclude`。如果`exclude`被设置为`true`,GridLayout将按照AbsoluteLayout的方式来安排它,也就是按照控件`setBounds()`方法设定的值来确定其位置。只有当`exclude`被设置成`false`时,GridLayout才会按照上面讲到的原则来确定这个控件的位置。`exclude`为`true`的控件的尺寸和位置也无法随容器的尺寸变化而变化,因此一般情况下不应该将这个值设为`true`。

5.6.5 FormLayout布局

在FormLayout中,某个控件的位置是由它与其他控件的相对位置来描述的。比如“Button A的上边界处于父容器高度的20%处”或“Button A的左边界处于Button B 右边界右方20像素处”,当容器尺寸变化时,其中的控件会根据这些信息调整自己的位置。

使用FormLayout时, 要为每一个控件创建一个FormData对象, FormData中有4个属性top, bottom, left和right, 代表了所对应控件的4个边界, 它们的类型都是FormAttachment, FormAttachment对象中包含了作为参照物的目标控件、所参照的边界以及相对位置等信息, 使用不同的信息创建FormAttachment并赋给FormData的对应属性, 就可以将控件边界绑定到所参照控件的相应位置。下面的代码创建了一个使用FormLayout的Shell (代码见光盘: \book.ch5.UsingFormLayout.java), 效果如图5-27所示。

```

.....

Display display = Display.getDefault();
Shell shell = new Shell(display);
shell.setLayout(new FormLayout());
shell.setSize(200,200);
shell.open();

// 为ButtonA 绑定边界
final Button buttonA = new Button(shell, SWT.NONE);
buttonA.setText("Button A");

FormData buttonDataA = new FormData();

// 左边界绑定到父控件宽度20% 向右偏移1像素处 (偏移为负则向左偏移)
buttonDataA.left = new FormAttachment(20,1);

// 右边界绑定到父控件宽度50%处
buttonDataA.right = new FormAttachment(50);

// 为ButtonB 绑定边界
buttonDataA.top = new FormAttachment(0); // 上边界绑定到父控件高度为0%处
buttonDataA.bottom = new FormAttachment(50); // 下边界绑定到父控件高度为50%处
buttonA.setLayoutData(buttonDataA);

final Button buttonB = new Button(shell, SWT.NONE);

// 左边界绑定到ButtonA右边界, 偏移为0
final FormData buttonDataB = new FormData();
buttonDataB.top = new FormAttachment(buttonA); // 上边界绑定到ButtonA下边界上 (未指定参照边界时, 水平默认为右边界, 竖直默认为下边界)
buttonDataB.left = new FormAttachment(buttonA,0,SWT.RIGHT);
buttonDataB.right = new FormAttachment(buttonA,60,SWT.RIGHT);
buttonDataB.bottom = new FormAttachment(buttonA,20,SWT.BOTTOM); // 右边界绑定到ButtonA右边界, 偏移为60像素
buttonB.setLayoutData(buttonDataB);
buttonB.setText("Button B");

shell.layout();

.....

```



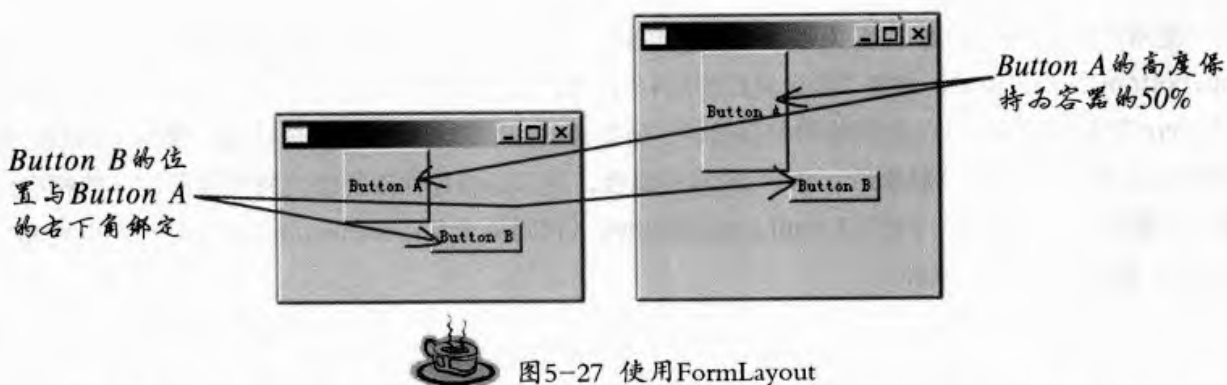


图5-27 使用FormLayout

上面的代码中首先为第一个Button对象创建了一个FormData对象，然后为它的4个边界用FormAttachment赋值。第一个参数代表本控件的边界相当于父容器尺寸的百分比位置，取值为0~100，第二个参数为在第一个参数基础上的偏移量，以像素为单位，取值可正可负。因此 `buttonData.left = new FormAttachment(20,1)` 就代表Button的左边界在Shell宽度的20%向右偏移1像素的位置上。如果Shell的当前宽度为100像素，Button的左边界就处在21像素处，第二个参数也可以省略。所以这段代码将Button A的左边界设为Shell宽度的20%+1像素，右边界设为Shell宽度的50%处；上下边界分别设为Shell高度的0%和50%处。

除了以父容器作为参照外，FormData也可以以其他兄弟控件作为参照，在代码中第二个Button对象的位置就是以第一个Button为参照设置的。与参照父控件的情况不同，参照兄弟控件时，是不能使用比例位置的，只能指定参照的边界和偏移量。默认情况下，水平方向以参照控件的右边界为参照，竖直方向以下边界为参照。因此代码 `buttonDataB.top = new FormAttachment(buttonA)` 就指定了Button B的上边界与Button A的下边界对齐，而 `buttonDataB.bottom = new FormAttachment(buttonA, 20, SWT.BOTTOM)` 则将Button B的下边界设为Button A的下边界+20像素。

※ 注意: ※

在使用FormLayout时，是不允许循环引用的。如果控件B引用控件A作为参照，那么所有以控件B作为参照的控件，包括控件B自身，都不能再作为控件A的参照。

5.6.6 StackLayout布局

之前的所有布局管理器都致力于在界面上同时排布所有的子控件，但是StackLayout却不是这样，在StackLayout管理的所有子控件中，只有一个能够显示出来，这个控件会占据父容器的所有显示区域，而其他控件都处于不可见状态，在这种布局管理下的所有子控件就好像是放在一个栈中，只有栈顶的控件才能显示，如图5-28所示。

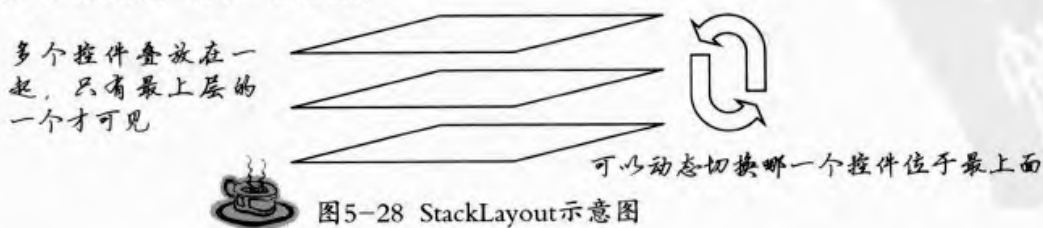


图5-28 StackLayout示意图

StackLayout通过属性topControl指定哪一个子控件当前处于栈顶。通过重新设置这个值，再调用Composite的layout进行重新布局，就可以切换处于可见状态的子控件。下面的例子在一个使用了StackLayout的容器中放置了10个Button，并通过单击一个控制按钮来依次切换处于顶部（可见状态）的控件(代码见光盘：\book.ch5.UsingStackLayout.java)，效果如图5-29所示。

.....

```

Display display = new Display();
Shell shell = new Shell(display);
shell.setLayout(new GridLayout());

final Composite parent = new Composite(shell, SWT.NONE);
parent.setLayoutData(new GridData(GridData.FILL_BOTH));
final StackLayout layout = new StackLayout();
parent.setLayout(layout);
final Button[] bArray = new Button[ 10];
for (int i = 0; i < 10; i++) {
    bArray[i] = new Button(parent, SWT.PUSH);
    bArray[i].setText("Button "+i);
}
layout.topControl = bArray[ 0];

Button b = new Button(shell, SWT.PUSH);
b.setText("Show Next Button");
final int[] index = new int[ 1];
b.addListener(SWT.Selection, new Listener(){
    public void handleEvent(Event e) {
        index[ 0] = (index[ 0] + 1) % 10;
        layout.topControl = bArray[ index[ 0]];
        parent.layout();
    }
});

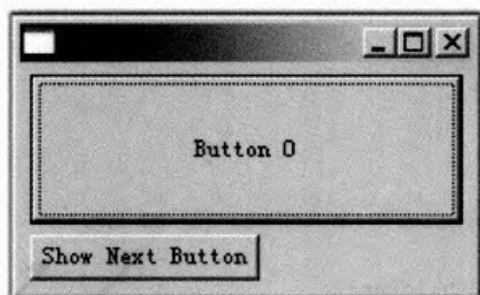
shell.open();

```

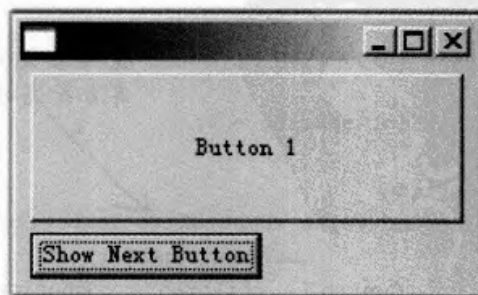
.....

创建10个重叠的
Button并将Button 0
放在最上面

当单击按钮时，将topControl设
为下一个button并调用layout方法
重新排布（必须调用layout方法
才能使改动生效）



第一个Button处于栈顶（可见状态）



第二个Button处于栈顶（可见状态）



图5-29 使用StackLayout

这种布局通常用于需要根据用户的选择动态改变应用程序界面的情况下。将每一种可能的界面安排在一个Composite中，然后在主界面中使用StackLayout对这些可能的Composite进行布局。当用户选择改变时，选择新的Composite作为顶部元素然后重新布局，就可以方便地改变界面。

StackLayout与第4章介绍的TabFolder比较类似，都是在同一个位置上安排不同的控件，并可以动态地切换它们，但是StackLayout不像TabFolder会显示一个选项卡，用户无法手动切换控件，使用时可以根据需要自由选择。

5.6.7 布局的选择规则

一般来说，可以将界面设计分为两个阶段，第一阶段是用容器类控件对窗口区域进行大概划分的过程，可以叫做“圈地”，第二阶段则是针对每个容器的用途，安排其中的具体控件位置。

在第一阶段，通常可以使用StackLayout和FillLayout。

在第二阶段中，GridLayout可以满足绝大部分的需求，如果有特殊需要，可以尝试使用RowLayout和FormLayout等其他布局。

AbsoluteLayout一般只适合使用在窗口尺寸预先规定好，而且无法变化的情况下，如对话框中。

5.6.8 自定义布局管理器

如果SWT自带的布局管理器不能满足需要，还可以自定义布局管理器来完全操纵控件的排布，下面就以一个例子来讲解如何自定义布局管理器。

所要创建的布局管理器描述为：控件在容器中竖直排列成一排，每个控件的宽度都等于容器宽度，高度占容器高度的比例由一个0~100的比例参数(ratio)指定，用户还可以指定页边间距(margin)和相邻控件间的间距(spacing)。

自定义的布局管理器名为“MyLayout”，其代码如下所示(代码见光盘：\book.ch5.MyLayout.java)，margin和spacing两个属性分别指定了页面间距和控件间距的值。

```
public class MyLayout extends Layout {  
  
    public int margin = 0;    该方法用于供容器计算自身偏好尺寸  
  
    public int spacing = 1;  
  
    @Override  
    protected Point computeSize(Composite composite, int wHint, int hHint,  
                                boolean flushCache) {  
        Control[] children = composite.getChildren();  
        int width = 0;  
        int height = 0;  
  
        for (int i = 0; i < children.length; i++) {  
            Control control = children[i];  

```

```

        Point size = control.computeSize(wHint, hHint);
        MyData data = (MyData) control.getLayoutData();

        width = Math.max(width, size.x);
        height = Math.max(height, size.y * 100 / data.ratio);
    }

    return new Point(width + 2 * margin, height + 2 * margin + (children.length - 1) * spacing);
}

```

该方法用于排布控件，容器的layout方法被调用时，就会调用布局管理器的layout方法

@Override

```

protected void layout(Composite composite, boolean flushCache) {
    Control[] children = composite.getChildren();
    Rectangle clientArea = composite.getClientArea();
    int width = clientArea.width - 2 * margin;
    int height = clientArea.height - 2 * margin - spacing * (children.length - 1);
    Point current = new Point(clientArea.x + margin, clientArea.y + margin);
    for (int i = 0; i < children.length; i++) {
        Control control = children[i];
        MyData data = (MyData) control.getLayoutData();
        int currentWidth = width;
        int currentHeight = height * data.ratio / 100;
        control.setBounds(current.x, current.y, currentWidth, currentHeight);
        current.y += currentHeight + spacing;
    }
}

```

MyLayout的布局信息类是MyData，它只有一个参数ratio，标明了控件在容器中的高度比例，代码如下所示(代码见光盘：\book.ch5.MyData.java)。

```

public class MyData {
    public int ratio = 10;
}

```

MyLayout中实现了Layout的两个抽象方法computeSize和layout，这也是所有布局管理器都必须实现的方法。

computeSize是容器用来计算自己的偏好尺寸的。前面讲到，所有控件都有自己的偏好尺寸，容器当然也不例外，既然控件在容器内的排布是由布局管理器控制的，那么容器的偏好尺寸自然也是和布局管理器息息相关，容器需要得到自身的偏好尺寸时，就会在它的布局管理器上调用computeSize。方法实现如下所示。

```

protected Point computeSize(Composite composite, int wHint, int hHint,
    boolean flushCache) {
    Control[] children = composite.getChildren();
    int width = 0;
    int height = 0;

    for (int i = 0; i < children.length; i++) {
        Control control = children[i];
    }
}

```



```

        Point size = control.computeSize(wHint, hHint); // 得到容器中每一个
                                                         控件的偏好尺寸
        MyData data = (MyData) control.getLayoutData();

        width = Math.max(width, size.x);
        height = Math.max(height, size.y * 100 / data.ratio);

    }
    return new Point(width + 2 * margin, height + 2 * margin +
        (children.length - 1) * spacing);
}

```

从控件上取得布局信息

根据布局信息计算容器的控件所需要的最小尺寸

实现逻辑不是很复杂，首先得到了容器的所有子控件(children)，然后遍历它们取得每个子控件的偏好尺寸。以其中的宽度最大值为容器的宽度，以其中高度/ratio的最大值作为容器的高度，最后再加上页面间距和控件间距后就得到了容器的偏好尺寸。

computeSize方法共有4个参数，其中Composite参数就是布局管理器所在的容器控件，而wHint和hHint两个参数则可以用来限制布局管理器所能够使用的空间尺寸，如果这两个值取为SWT.DEFAULT，则代表没有限制。举例来说，如果容器控件的宽度为150，而wHint取值为100，则相当于通知布局管理器所能使用来排布控件的尺寸最多为100像素。这时布局管理器就要负责在这种限制下重新计算宽度和高度；而如果wHint取值为SWT.DEFAULT，则代表布局管理器可以使用全部150像素的空间，在MyLayout中，并未处理wHint和hHint这两个参数。最后一个参数flushCache则通知布局管理器是否要刷新其维护的、与容器相关的用于计算布局的信息，如果布局管理器中没有缓存这些信息，则无须理会这个参数。

layout方法就是在调用shell.layout()时，容器类调用的方法。这个方法真正负责着排布控件的任务，实现代码如下所示。

```

protected void layout(Composite composite, boolean flushCache) {
    Control[] children = composite.getChildren();
    Rectangle clientArea = composite.getClientArea();
    int width = clientArea.width - 2 * margin;
    int height = clientArea.height - 2 * margin - spacing * (children.length - 1);
    Point current = new Point(clientArea.x + margin, clientArea.y + margin);
    for (int i = 0; i < children.length; i++) {
        Control control = children[i];

        MyData data = (MyData) control.getLayoutData();

        int currentWidth = width;

        int currentHeight = height * data.ratio / 100;

        control.setBounds(current.x, current.y, currentWidth, currentHeight);

        current.y += currentHeight + spacing;
    }
}

```

根据布局算法计算每一个控件所占用的空间，并用setBounds方法对控件尺寸进行设置

在调用Composite的getClientArea方法取得容器的当前客户区域（客户区域是容器中去掉边界区域后真正可以供子控件显示的部分）尺寸后，开始从上到下计算每一个控件所占的位置和尺寸并调用setBounds来设置它们。当每一次容器尺寸变化时，layout方法都会被调用以重新计算各个控件的位置。

下面是一个使用自定义布局的示例，在示例中，创建了一个Label，一个Text和一个Button，并设定它们的比例参数分别是20%，30%和50%；页面间距和控件间距分别设成2和10像素(代码见光盘：\book.ch5.UsingMyLayout.java)，效果如图5-30所示。

.....

```
Display display = Display.getDefault();
Shell shell = new Shell(display);
shell.setText("My Layout");
MyLayout layout = new MyLayout();
layout.margin = 2;
layout.spacing = 10;
shell.setLayout(layout);
shell.setSize(200,200);
```

```
Label label = new Label(shell,SWT.BORDER);
label.setText("This is a Label");
MyData labelData = new MyData();
labelData.ratio = 20;
label.setLayoutData(labelData);
```

```
Text text = new Text(shell,SWT.BORDER);
text.setText("This is a text");
MyData textData = new MyData();
textData.ratio = 30;
text.setLayoutData(textData);
```

```
Button button = new Button(shell,SWT.NONE);
button.setText("This is a button");
MyData buttonData = new MyData();
buttonData.ratio = 50;
button.setLayoutData(buttonData);
```

```
shell.open();
shell.layout();
```

.....

使用自定义Layout的步骤与使用SWT自带的Layout相似，创建一个布局信息类，设置其属性并赋到控件上。这个信息类会由布局管理器读取



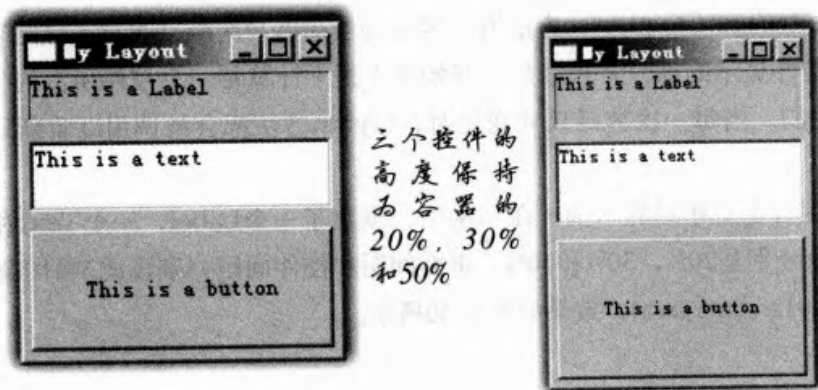


图5-30 使用自定义的Layout

这样，就成功地得到了一个自己的布局管理器，从而也对布局管理器的工作原理有了更深入的认识。

5.7 本章小结

本章主要讲述了容器类控件的使用，并对各种布局类以及如何自定义布局作了较为深入的讲解。各种布局类的灵活使用是创建美观大方界面的基础，读者需要在实践中进一步加深理解。下一章中，将讲述如何使用可视化的开发工具来简化界面开发。

我们通过实践证明，
得出结论：我掌握了
如此简单的方法！



第6章 界面开发工具

前面的学习过程中，都是直接编写代码进行界面设计。进行图形界面设计时，如果有一个“所见即所得”的工具帮助，可以大大提高工作效率，本章将介绍一个这样的工具—Visual Editor。通过本章的学习，读者将会熟悉如何使用Visual Editor进行常用的SWT界面开发工作。

本章内容包括：

- ★安装Visual Editor。
- ★使用Visual Editor。
- ★其他工具介绍。

拉开崭新的学习帷幕

进入第06章



6.1 安装Visual Editor

Visual Editor(简称VE)是隶属于Eclipse社区的一个开源项目, 它的最新版本可以在<http://download.eclipse.org/tools/ve/downloads/index.php>下载, 下载时有几点需要注意, 首先, Visual Editor是以EMF和GEF为基础开发的。因此运行Visual Editor之前需要先将这两个插件安装好, 不同的Visual Editor版本对Eclipse IDE、EMF与GEF的版本需求也是不同的, 具体信息在每一个版本的下载说明中都有详尽的介绍。

Visual Editor的下载页面包含了Runtime和SDK两种类型, 如果只需要使用Visual Editor进行界面设计, 使用Runtime就足够了; 如果希望对Visual Editor进行二次开发, 就需要下载SDK, 它除Runtime的内容外, 还包含了源代码和说明文档。另外, 页面中还包含有Visual Editor的示例和基于JUnit的测试用例下载, 以下的说明均以Runtime为基础。

下载Runtime的压缩包后, 将它解压到Eclipse的安装目录下; 然后重新启动Eclipse IDE。打开Window→Preferences对话框, 如果在Java类别下面多出一项“Visual Editor”, 如图6-1所示, 就说明安装已经成功了。

找到这一项,
说明安装成功

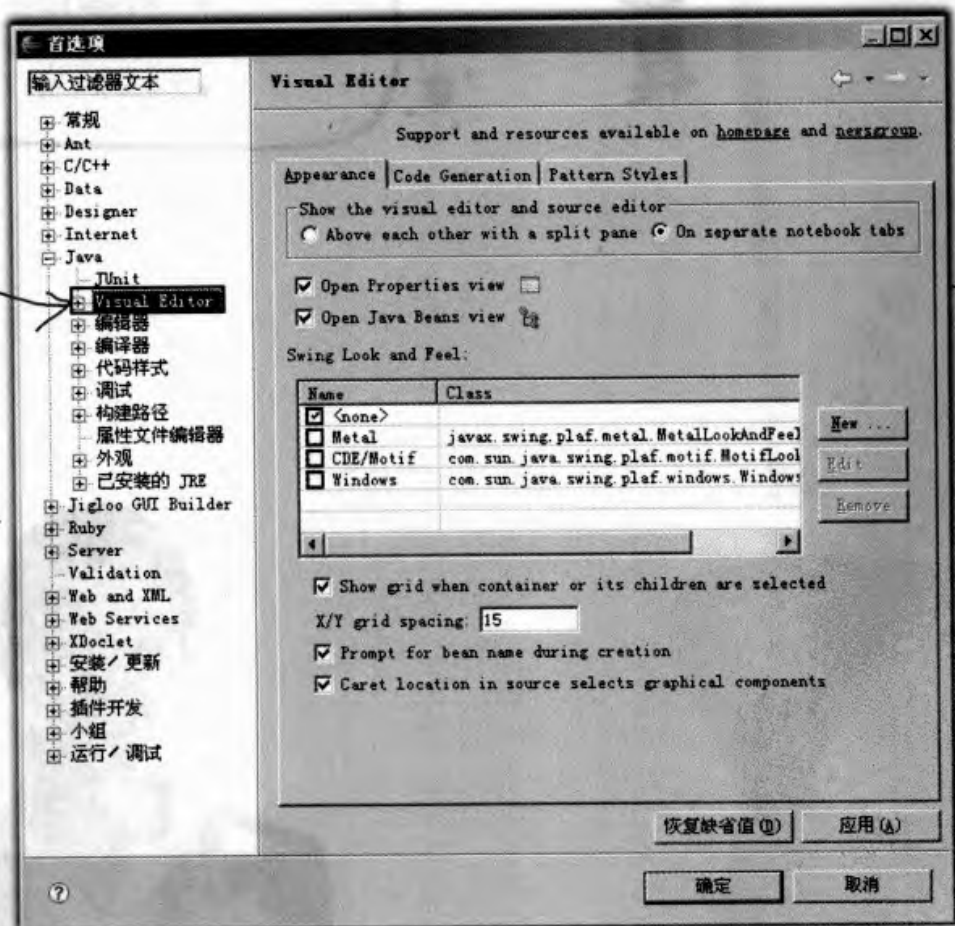


图6-1 Visual Editor的设置界面

6.2 使用Visual Editor

6.2.1 Visual Editor的基本使用

首先从创建一个Shell开始，演示如何使用Visual Editor进行界面开发。单击菜单“文件”→“新建”→“其他”，打开“新建”对话框。在Java项目下面可以看到一个类型“Visual Class”，这代表了Visual Editor的文件类型。选择它并单击“下一步”按钮。

在下一个页面中，如图6-2所示，可以选择希望创建的界面的类型。从选项中可以看到，Visual Editor支持基于SWT/Swing/AWT的图形界面。在Style中选择SWT→Shell，输入类名后单击“完成”按钮，Visual Editor会生成代码并打开一个可视化编辑界面。

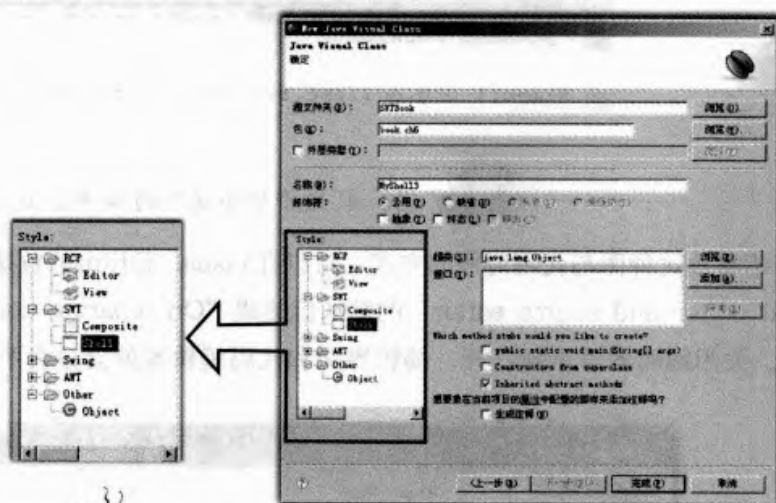


图6-2 选择Visual Class类型

如图6-3所示，Visual Editor的编辑界面分为上下两个部分。上面的设计视图以图形化的方式显示用户正在设计的图形界面，下面的代码视图则显示着对应的Java代码。两个视图是互动的，在设计界面上添加一个控件时，代码会随之更新，反之亦然。用鼠标选中一个控件，编辑器中对应的代码（构造函数部分）会突出显示，鼠标定位到编辑器中代表某个控件的变量时，设计界面中对应的控件也会处于选中状态。

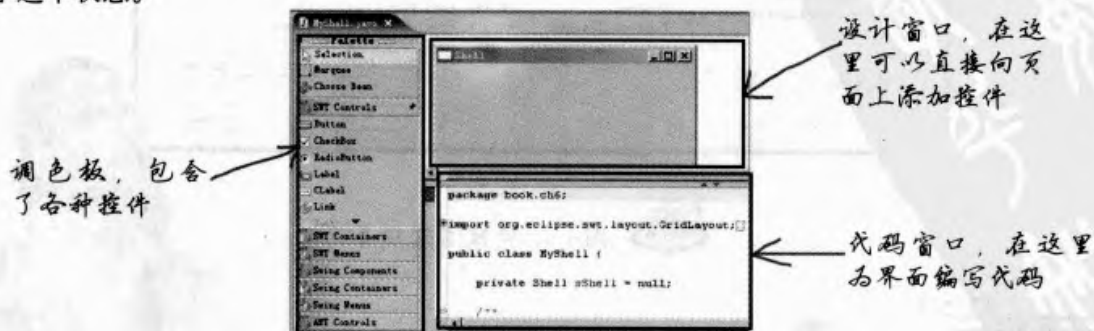


图6-3 设计视图与代码视图

默认情况下,两个界面是同步刷新的。如果由于性能等原因希望暂停自动刷新而改用手动刷新时,就要用到VE在工具栏上添加的暂停按钮。单击主工具栏上的暂停(Pause)按钮,设计界面会显示“Visual Editor paused”的字样,如图6-4所示。暂停按钮也会变成刷新(Reload)的样子。这时只能修改代码部分,界面不会随代码刷新。单击刷新按钮时,VE会重新分析代码并更新设计界面的内容。

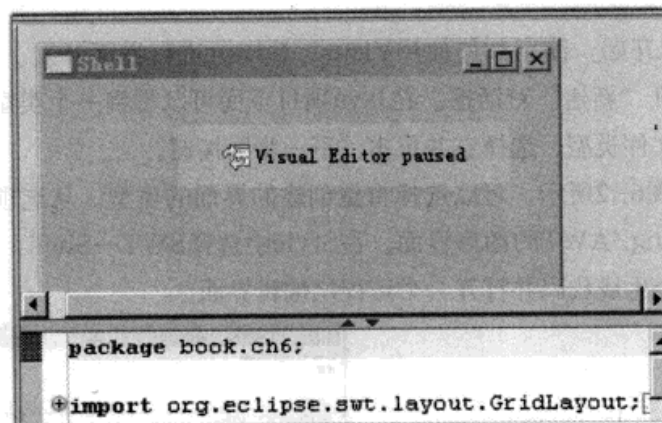


图6-4 处于暂停状态的编辑界面

如果不喜欢这种图与代码对照的方式,可以在Visual Editor的首选项页面中将“Show the visual editor and source editor”的默认值改成“On separate notebook tabs”,如图6-5所示。关闭编辑器再重新打开,编辑界面和代码编辑器就会分在两个页面中,如图6-6所示。

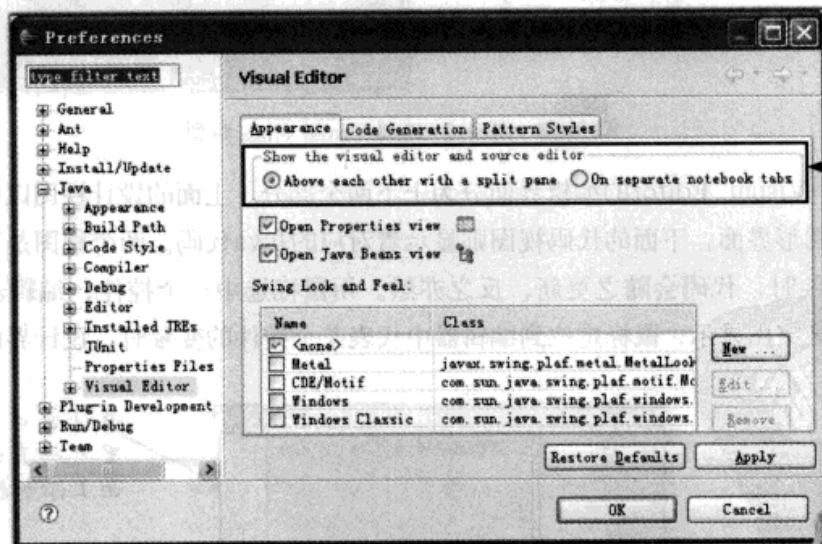


图6-5 修改首选项界面

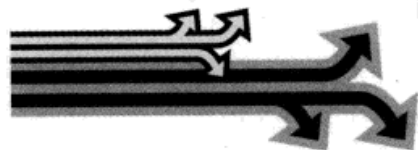




图6-6 把设计界面样式改成分页模式

在界面的左方，有一个可以隐藏或展开的控件工具箱(Palette)，其中包含了选择工具和许多组件按钮，任选一个控件，将鼠标移动到Shell上，Visual Editor会用一个位置标记显示这个控件可以摆放的位置，单击就可以在界面上加入这个控件，如图6-7所示。

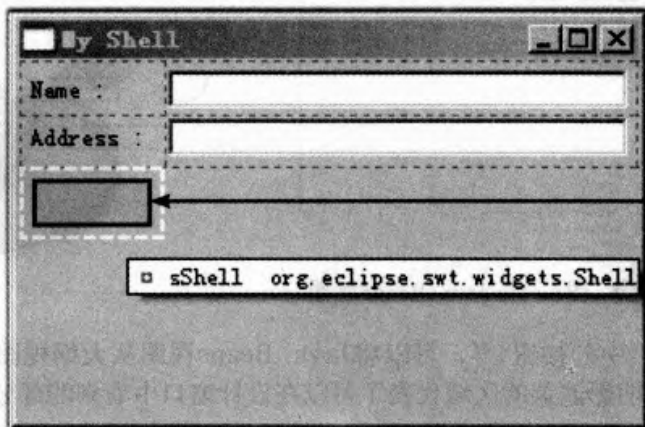


图 6-7 Shell中显示可以摆放控件的位置

控件工具箱包含了三种选择工具，选择、框选和选择Bean，如图6-8所示。



图6-8 三种选择工具

其中“选择Bean”一项允许用户 VE 编辑器中使用自己的 Java Bean，Java Bean 是指独立的、可重用的 Java 组件。以第 5 章中编写的“MyControl”为例，单击 Choose Bean，在弹出的选择框中选择 book.ch5.MyControl，然后在设计界面上单击鼠标，MyControl 就出现了，如图 6-9 所示。

使用Select Bean工具，可以将自定义的控件添加到Visual Editor的设计界面上

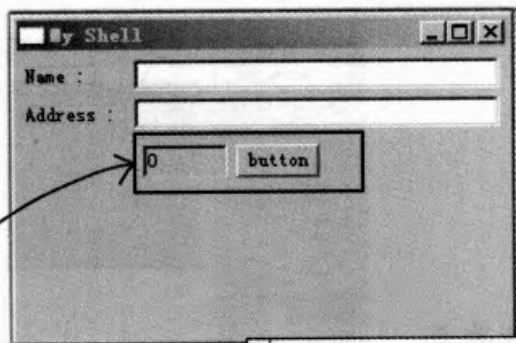


图6-9 向界面上添加JavaBean

Visual Editor提供了一个名为“Java Beans”的视图，如图6-10所示，它显示了设计窗口中所有控件，以及它们的层次结构。单击控件按钮再单击该视图窗口中的对象，也可以添加一个控件到界面上。

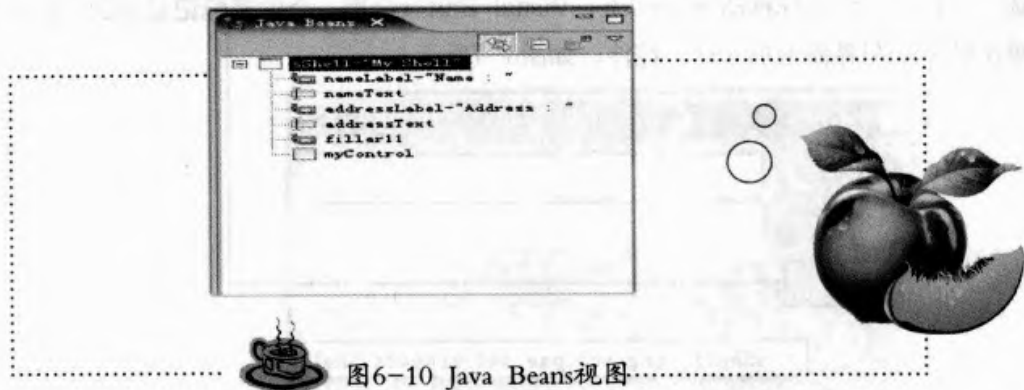


图6-10 Java Beans视图


单击Java Beans视图工具栏中的按钮, 可以将Java Beans视图从大纲视图切换到缩略图模式，如图6-11所示。缩略图中被阴影遮盖的区域代表了可以在设计窗口中看到的部分，鼠标拖动这个区域就可以方便地将设计窗口移到需要编辑的部分去添加、调整或删除控件。这在设计较大界面时很有用。



图6-11 缩略图模式的Java Beans视图

右键单击控件，在弹出菜单中选择“Events”，可以方便地为控件添加各种事件监听器。在菜单中直接列出的是所选择控件特有的事件类型。选择“Add Events”会列出控件支持的所有事件，如图6-12和6-13所示。

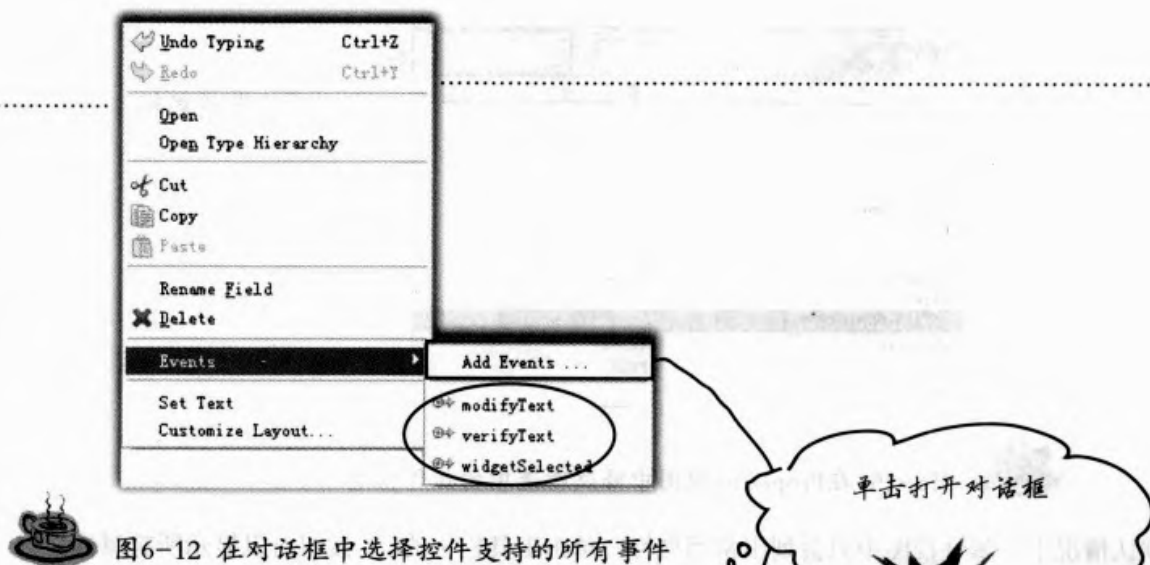


图6-12 在对话框中选择控件支持的所有事件

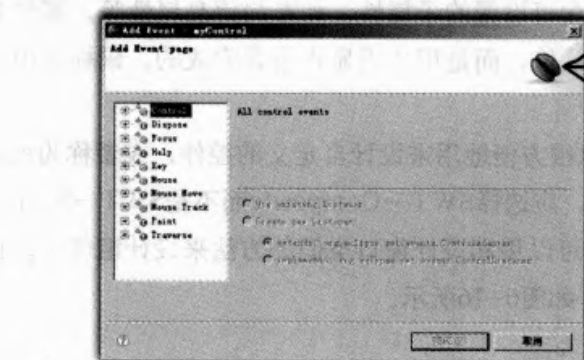


图6-13 在对话框中选择控件支持的所有事件

在Java Beans视图中，可以方便地看到当前哪些控件被安装了监听器，如图6-14所示。



图6-14 在Java Beans视图中观察事件监听器

在属性视图，可以编辑当前界面中选中的控件的各种属性。这些改动也会即时在界面和代码中反映出来，如图6-15所示。

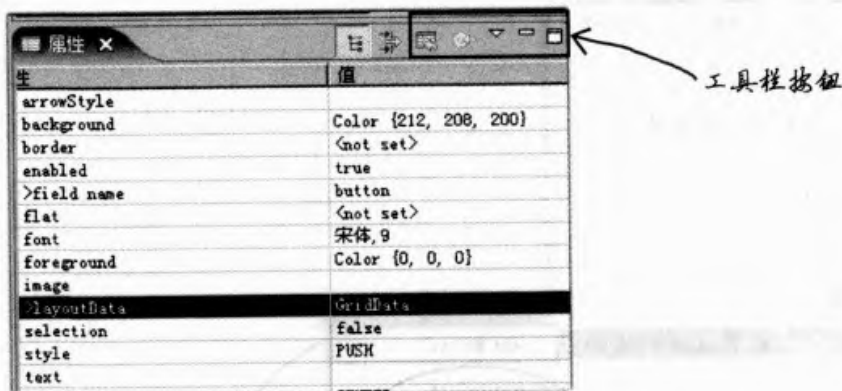


图6-15 在Properties视图中修改所选中的控件信息

默认情况下, 属性视图中只会列出常用属性, 单击工具栏上的 按钮可以显示所有属性。

属性名前面的加号(+)代表这个属性本身包含可设置的子属性, 如布局信息属性等。属性名称前面的右尖括号(>)代表这个属性的设置不是取默认值, 而是用代码显式设置完成的。鼠标选中这样的属性时, 对应的代码也会高亮显示。

除了直接设计窗口外, Visual Editor也可以很方便地用来设计自定义的控件, 或者称为可视化组件(Visual Java Bean)。在创建Visual Class时, 应选择SWT→Composite而不是SWT→Shell, 就可以创建一个继承自Composite类的组件。随后就可以用和设计窗口类似的方法来设计组件了。创建好组件后, 可以用“选择Bean”工具来使用它们, 如图6-16所示。

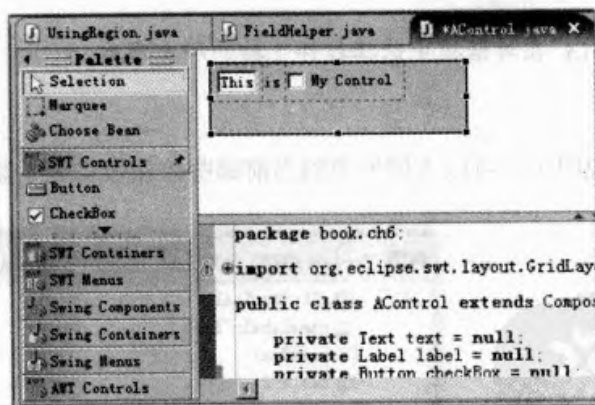


图6-16 使用Visual Editor设计可视化组件

6.2.2 控件布局

布局管理是界面设计的难点和关键, Visual Editor目前支持5种布局方式, null(Absolute) Layout, FillLayout, FormLayout, GridLayout和RowLayout, 自定义的布局管理器无法在Visual Editor中使用。

选中一个Shell，在属性视图的layout属性中会显示出它当前使用的布局管理器，默认情况下，新创建的容器控件都使用GridLayout。可以在“首选项”→“Java”→“Visual Editor”→“SWT”中改变这个默认值，如图6-17所示。

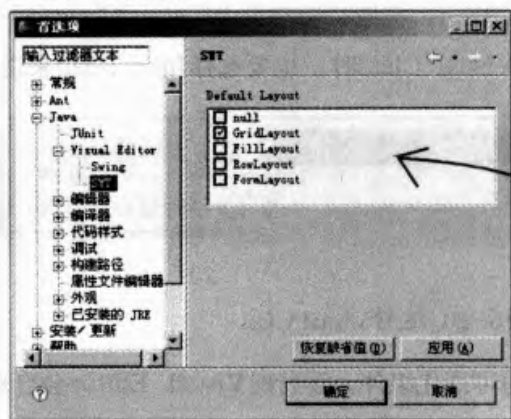


图6-17 默认布局设置

展开layout属性，可以设置GridLayout所特有的属性，如margin, spacing, numColumns等，另外，也可以右键单击容器，在弹出菜单中选择“Customize Layout”，会弹出如图6-18所示的对话框，在对话框中也可以方便地修改布局管理器的各项参数，这种方法比较适合初学者使用。

根据所选择布局管理器的不同，这个对话框的内容会随之变化

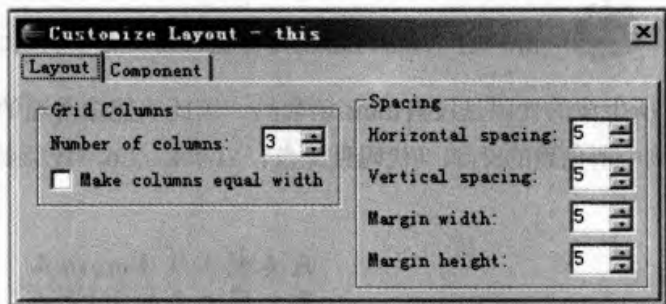


图6-18 自定义布局管理器属性的对话框

为某个容器选定了布局管理器后，容器中所有控件的layoutData属性都会被相应地改变成对应的布局信息类。选中某个控件，展开layoutData属性，就可以按照需要调整控件的布局信息了。在某个控件上右键菜单选择“Customize Layout”，会调出一个更为直观的属性编辑器，如图6-19所示，结合第5.6节中所讲述的布局基础知识，可以方便地设计自己喜欢的界面。

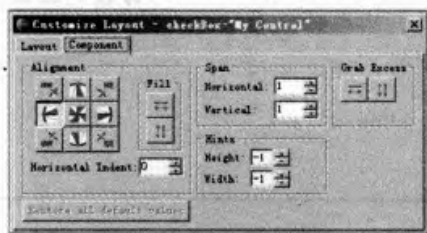


图6-19 控件的layoutData属性编辑器



6.2.3 运行与调试

在设计视图中虽然能够即时看到所设计的界面，但是很多逻辑还是要运行一下才能确认正确性。在Visual Editor中右键单击创建的Java Bean，选择“运行方式”→“Java Bean”，就可以运行程序了，如图6-20所示。同样地，在调试Visual Class时，也要选择Java Bean来进行调试。

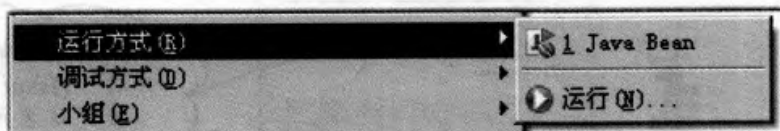


图6-20 运行Visual Class

如果创建的是一个基于Composite的可视化组件，运行时Visual Editor会自动创建一个Shell并将组件嵌入其中，这样就不需要单独编写一个类来测试组件了。如运行刚才创建的新控件，效果如图6-21所示。

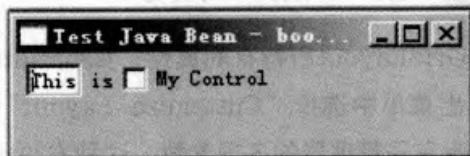


图6-21 Visual Editor为可视化组件自动创建测试窗口

Visual Editor所生成的代码是没有main函数的，因此不能像普通的Java程序一样运行。下面的代码是在之前的示例中所创建的Shell（代码见光盘：\book.ch6.MyShell.java）。

```
.....

public class MyShell {

    private Shell sShell = null; // @jve:decl-index=0:visual-
    constraint="10,10"
    private Label nameLabel = null;
    private Text nameText = null;
    private Label addressLabel = null;
    private Text addressText = null;
    private Button button = null;
    /**
     * This method initializes sShell
     */

    private void createSShell() {
        GridLayout gridLayout1 = new GridLayout();
        gridLayout1.numColumns = 2;
        sShell = new Shell();
    }
}
```

该类型中没有main函数，因此只能用VE的JavaBean方式运行

以“@jve”起始的注释是VE添加的控制信息，注意不要删除它们

```
sShell.setText("Shell");
sShell.setLayout(gridLayout1);
sShell.setSize(new Point(263, 191));
nameLabel = new Label(sShell, SWT.NONE);
nameLabel.setText("Name");
nameText = new Text(sShell, SWT.BORDER);
addressLabel = new Label(sShell, SWT.NONE);
addressLabel.setText("Address");
addressText = new Text(sShell, SWT.BORDER);
Label filler = new Label(sShell, SWT.NONE);
button = new Button(sShell, SWT.NONE);
button.setText("button");
button.addSelectionListener(new
org.eclipse.swt.events.SelectionAdapter() {
    public void
widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        System.out.println("widgetSelected()");
    }
});
}
```

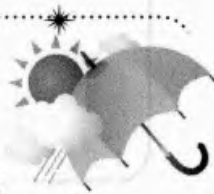
VE生成的代码，只有一个createSShell()的函数，在这个函数中包含了创建shell和其他控件的代码



以Java Bean模式运行时，程序是从Visual Editor内部的JavaBeansLauncher的main函数开始执行的。包括SWT程序的事件循环处理等通用的代码都被封装在用于运行基于SWT的Bean程序的SWTLauncher中。如果读者有兴趣的话，可以下载Visual Editor的SDK查看其源代码。

※ 注意 ※

用VE开发时，如果使用了图片或字体资源，VE只会生成创建Image/Font等资源对象的代码，而不会生成相对应的释放这些资源的代码。开发者需要自行完成这部分工作。



Visual Editor采取的这种方法降低了开发者的工作量和代码的复杂度，开发者可以真正地集中精力在程序的内部逻辑上，但是这样生成的代码离开了Visual Editor环境是无法运行的。如果在创建Visual Class时选择创建main函数，Visual Editor会在main函数中加入事件处理等内容的代码，如图6-22所示。这样的程序就可以直接通过“运行方式” → “SWT应用程序”来运行。下面的代码中包含了Visual Editor自动生成的main函数代码，其中包含了熟悉的创建Shell和事件循环等内容(代码见光盘：\book.ch6.MyShell2.java)。



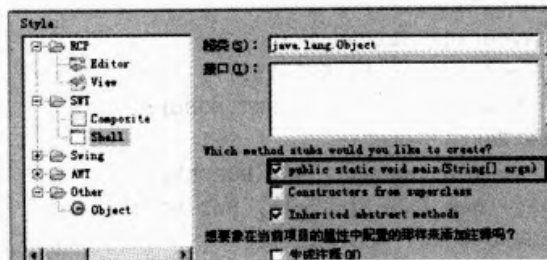


图6-22 在创建Visual Class时选择创建main函数

```
public class MyShell2 {
```

```
    private Shell sShell = null;
    private Label nameLabel = null;
    private Text nameText = null;
    private Label addressLabel = null;
    private Text addressText = null;
```

```
    /**
     * @param args
     */
```

```
    public static void main(String[] args) {
        Display display = Display.getDefault();
        MyShell2 thisClass = new MyShell2();
        thisClass.createSShell();
        thisClass.sShell.open();

        while (!thisClass.sShell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }
```

```
    /**
     * This method initializes sShell
     */
```

```
    private void createSShell() {
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 2;
        sShell = new Shell();
        sShell.setText("Shell");
        sShell.setLayout(gridLayout);
        sShell.setSize(new Point(300, 200));
        nameLabel = new Label(sShell, SWT.NONE);
        nameLabel.setText("Name");
        nameText = new Text(sShell, SWT.BORDER);
        addressLabel = new Label(sShell, SWT.NONE);
        addressLabel.setText("Address");
        addressText = new Text(sShell, SWT.BORDER);
    }
}
```

选择了生成main函数后，
VE会生成一个包含了消息
循环的main函数，这时就可
以通过正常方式运行它了

和前面的例子中一
样的内容，创建
Shell和控件

6.3 其他工具介绍

除了Visual Editor外,还有很多基于Eclipse平台的优秀的可视化界面设计工具。下面来看另外两种主流产品, SWT Designer和Jigloo。

6.3.1 SWT Designer

SWT Designer是Instantiation公司出品的一款商业级图形界面设计插件,曾获得2006年度“Eclipse最佳商业插件奖”的荣誉。在它的官方网站<http://www.swt-designer.com/>可以下载到最新版本的SWT Designer。Instantiation公司是一个老牌的Eclipse商业级插件提供商,在GUI工具开发方面拥有丰富的产品,其中比较著名的,除SWT Designer外,还有基于Google Web Toolkit的GWT Designer等产品。

与Visual Editor相同,SWT Designer也是基于GEF技术的。但相对而言,SWT Designer的功能更为全面和丰富。在界面响应速度和易用性上也要胜出一筹,并且支持JFace Viewer的使用。用SWT Designer生成的代码是不依赖于SWT之外任何环境的。图6-23显示了SWT Designer的设计界面。

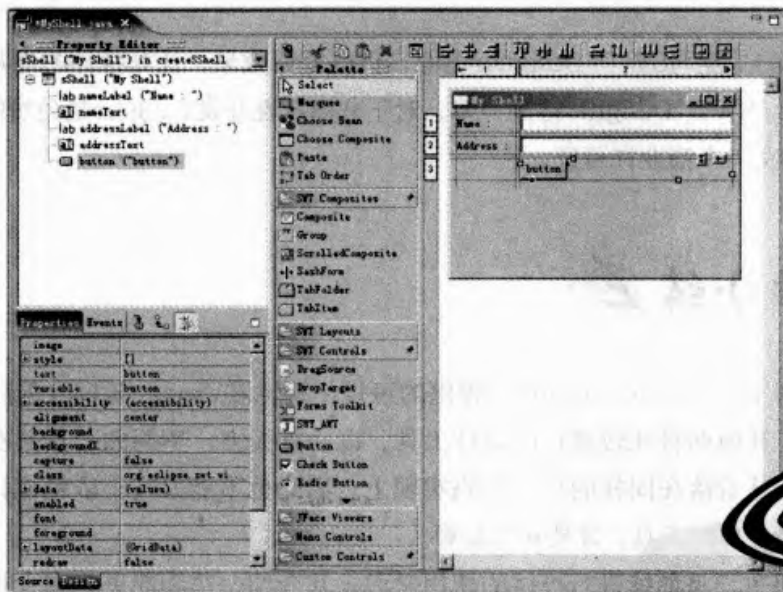


图6-2 SWT Designer的设计界面

作为一款商业软件,SWT Designer只提供三天的限制功能免费试用,超过期限后需要购买许可证才能继续使用。

6.3.2 Jigloo

Jigloo是由CloudGarden公司设计的一款用于Eclipse和IBM WebSphere Studio的Java GUI设计工具,其界面如图6-24所示。它的官方网站是<http://www.cloudgarden.com/jigloo/>。

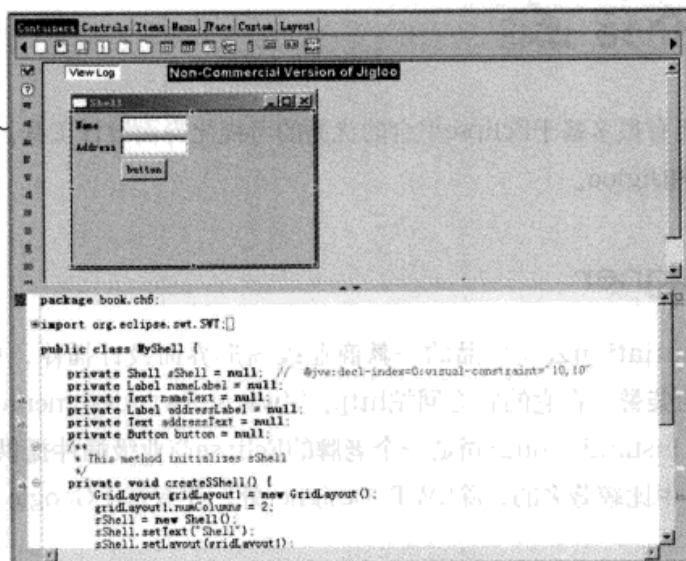


图6-24 Jigloo的界面



Jigloo所提供的功能与SWT Designer类似，但以界面效果，比如控件的布局定位等功能来说，要比Visual Editor/SWT Designer略逊一筹。对于非商业性开发，Jigloo是免费的，但如果要用于商业开发，就需要购买一个商业许可证。

6.4 本章小结

本章简单介绍了如何使用Eclipse社区提供的可视化编辑器Visual Editor进行SWT程序的界面设计，同时还介绍了其他两种比较流行的设计工具。以功能而言，不同的工具其实相差并不多，毕竟所有的设计最后还是着落在同样的Java代码实现上。因此也不能太过于依赖设计工具。在充分掌握SWT基本功的基础上使用工具，才能相得益彰。

在下面的章节中，将继续进行控件的使用学习，并介绍一些功能更为强大也更为复杂的控件，在这个过程中，会广泛使用Visual Editor进行开发，因此在进一步学习之前，读者有必要熟练掌握Visual Editor。

集体的力量是不容忽视的!



第7章 高级控件使用

在学习了基本控件的使用，并掌握了使用可视化工具设计界面后，本章将继续介绍如何使用一些高级控件，如Table、Tree等，并将剖析JFace的查看器框架，并通过实例演示如何使用查看器来简化使用复杂控件的步骤。通过学习本章，读者将对查看器框架有所了解，并能够使用列表查看器、表格查看器和树查看器进行开发工作。在本章的最后，将对SWT/JFace中其他一些功能强大的控件，包括StyledText、滚动条、进度条及OLE技术等作了概要的介绍。

本章内容包括：

- ★列表、表格和树。
- ★文本编辑器。
- ★滚动条、Scrollable、ScrolledComposite和滑动条。
- ★进度条与进度指示器。
- ★浏览器与OLE。

拉开崭新的学习帷幕

进入第07章



7.1 列表、表格和树

显示复杂的结构化数据类型时，仅仅使用简单控件是无法满足需求的。这时就需要使用列表、表格和树等高级控件；高级控件功能强大，但使用起来也相对复杂，用SWT代码控制它们是一项纷繁复杂的工作；为了简化这些工作，JFace为这些高级控件提供了查看器(Viewer)。本节在简介查看器框架后，将介绍如何使用纯SWT代码控制高级控件，随后再用查看器完成相同的工作，在对比中发现查看器的优点。

7.1.1 查看器框架

在查看器框架中，将模型称为输入(input)，查看器本身充当控制器的角色，而高级控件本身作为视图，当输入改变时，查看器负责相应地改变控件的内容。图7-1显示了JFace中的查看器结构。本节将主要介绍列表查看器、表格查看器和树查看器，它们分别对应到SWT控件List, Table和Tree。

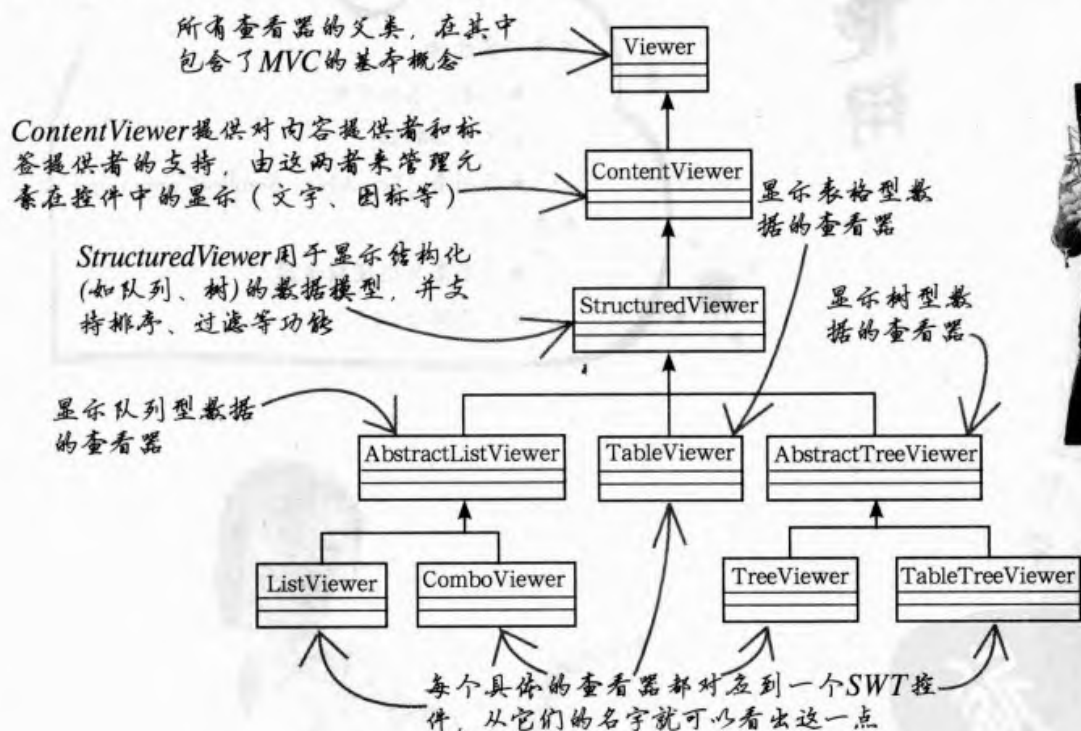


图7-1 查看器结构

首先需要明确，查看器可以做的事情，直接使用控件也完全可以做到，查看器只不过是简化了对复杂控件的操作而已。使用查看器，开发者只需要对数据模型进行操作就可以方便地改变控件的显示内容。

查看器的框架由以下几个主要部分构成。

★模型和元素——存储着要显示在控件中的数据模型（由开发者编写）；

★内容提供者和标签提供者——负责将数据模型转化成可以显示的图片 and 文字（由开发者编写）；

★控件——显示内容；

★查看器——负责协调上面各部分的工作。

查看器所使用的模型是任意的Java对象，如一个用来显示公司雇员列表的列表查看器中会维护许多雇员对象，那么一个包含所有雇员对象的队列（java.util.List）就可以作为列表查看器的模型使用。通常当模型的内容有所改变时，应该向外界发出一个“属性改变”的事件，这样查看器可以及时得到模型的改变信息从而刷新控件内容。

显示在查看器中的数据被称为元素，数据模型所维护的就是元素的集合，不同的查看器需要的元素集合类型也不同，如列表查看器和表格查看器显示的都是一行行的数据，它们需要的元素集合是一个对象数组；而树查看器显示的是树型结构，它需要的就是树型结构的元素集合。

多数情况下，外部程序无法直接提供查看器需要的模型，开发者需要自己解析模型并从中取得查看器需要的内容，这一步工作由内容提供者来完成。所有内容提供者都要实现接口IContentProvider，对应到具体的查看器子类，又有针对它们的数据结构的各种接口，如列表查看器和表格查看器要求内容提供者实现IStructuredContentProvider接口，而树查看器的内容提供者则需要实现ITreeContentProvider接口。如图7-2所示。

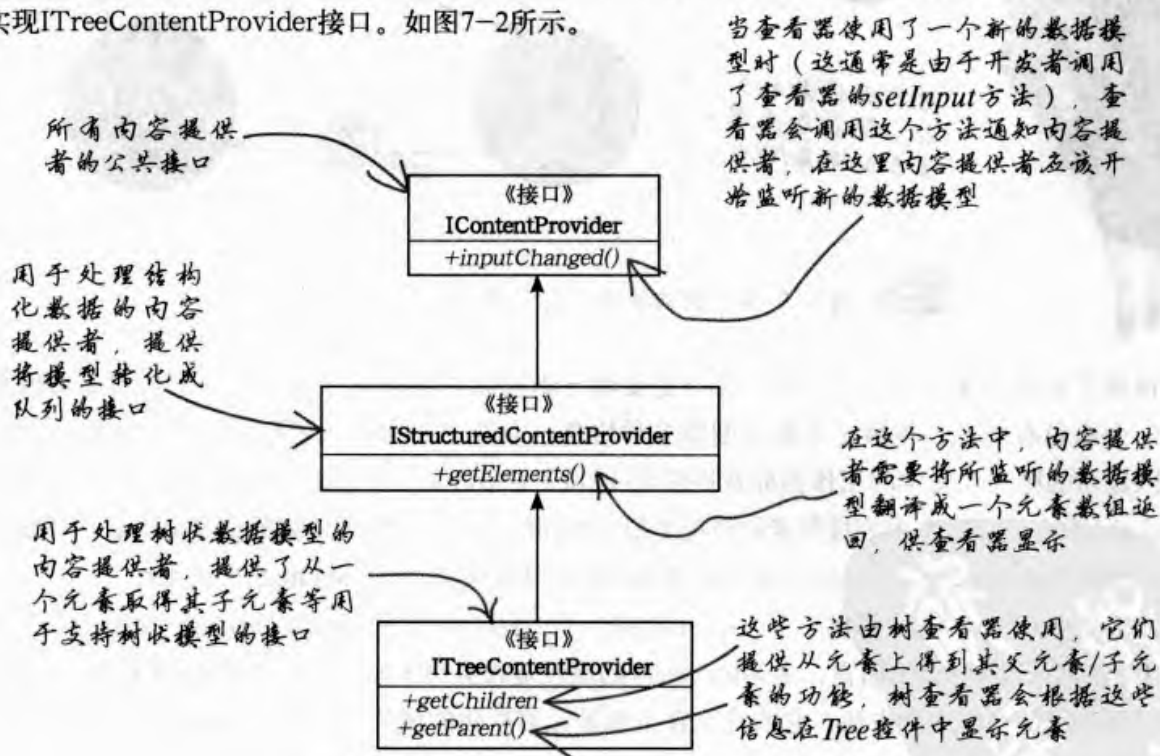


图7-2 内容提供者的类型层次结构

内容提供者的另外一项工作就是监听数据模型的变化，当接收到变化的消息时，要操作查看器以通知它这些变化，查看器会把这些操作转化成针对控件的操作并将变化反映在控件的界面上。表7-1中列出了部分常用的操作查看器的方法，另外不同的查看器还提供各自特有的一些操作方法，这在讲到具体查看器时会加以说明。图7-3演示了变化发生时，各部分是如何互动的。

表7-1 操作查看器的方法

方法名	方法用途
refresh()	刷新整个查看器的内容, 查看器会重新从数据模型中读取全部内容并刷新控件
refresh(Object element)	刷新指定的元素, 如果这个元素有子结构(如树状结构中的一个父节点), 它的所有子结构会一起被刷新
update(Object element, String[] properties)	更新指定的元素, 与refresh方法不同, update方法只会更新元素本身的显示内容, 而不会更新该元素的子结构。properties参数用于通知查看器该元素中有哪些属性发生了变化, 查看器可以据此决定是否需要更新这个元素

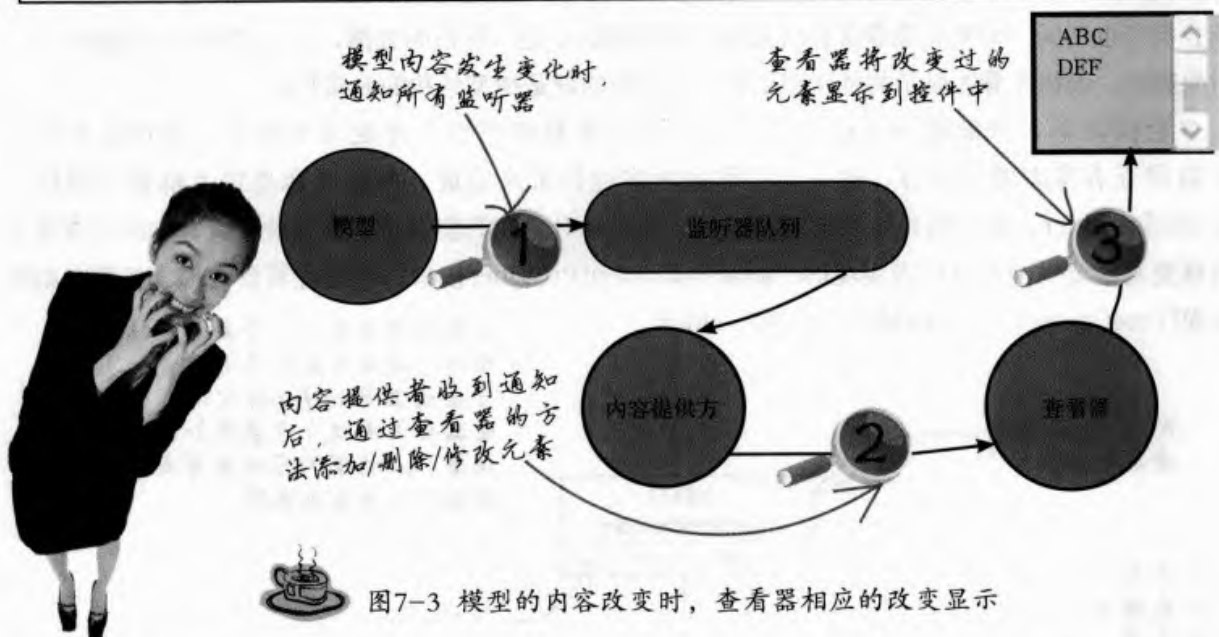


图7-3 模型的内容改变时, 查看器相应的改变显示

得到了元素的集合后, 下一步工作就是要将元素显示在界面上。仍然以显示公司雇员信息为例, 一个雇员有姓名、年龄、入职日期等多种信息, 或许还要为雇员显示照片, 这些将元素转化成可以显示的图片 and 文字的工作由标签提供者(Label Provider)来完成。标签提供者需要实现接口 IBaseLabelProvider。与内容提供者的情况类似, 不同的查看器也需要不同的标签提供者, 列表查看器可以使用子接口 ILabelProvider, 而表格查看器则必须使用接口 ITableLabelProvider。

IBaseLabelProvider中提供一个方法 isLabelProperty(Object element, String property), 该方法用于判定对元素element而言, 名为property的属性是否会用于显示。在更新元素的时候, 查看器会调用标签提供方的这个方法, 它的意义在于如果元素变化的属性不会影响到显示的内容, 则没有必要更新这个元素的显示。这也有利于提高程序的执行效率。查看器在执行update方法时, 就是利用 IBaseLabelProvider的这个方法来判定该元素是否需要更新的。

图7-4演示了查看器框架中各部分之间是如何协同工作的。



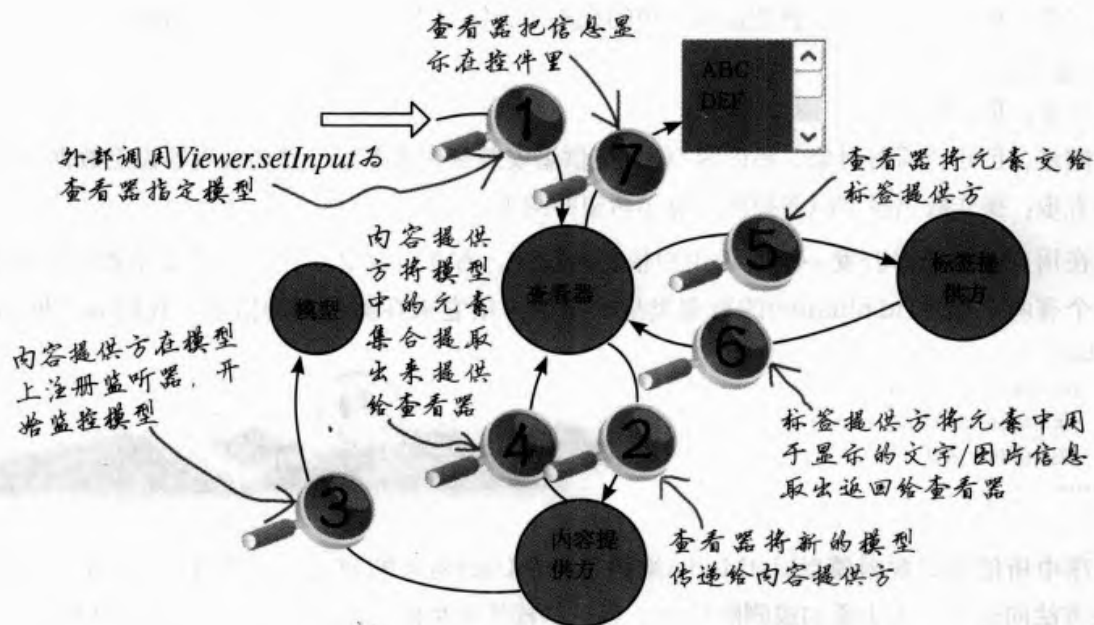


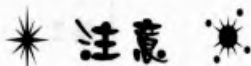
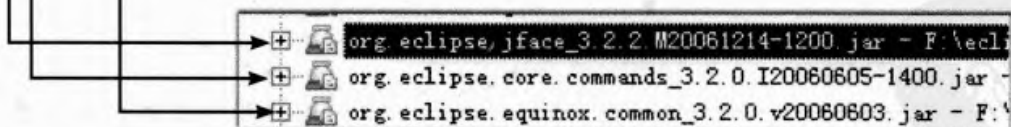
图7-4 查看器框架中各对象之间的关系

7.1.2 JFace 列表查看器(List Viewer)

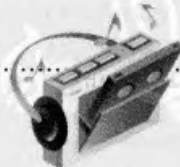
在使用查看器之前, 需要将JFace的JAR包加到Eclipse的编译路径里面, 表7-2列出了JFace所需要的JAR包, 在{eclipse安装目录}/plugin中, 可以找到它们。当然, 还要确保SWT的JAR已经配置好。

表7-2 使用JFace所需要的JAR包

包名	包内容
org.eclipse.jface	包含JFace的基本类型
org.eclipse.core.commands	包含JFace所用到的部分Eclipse平台的代码
org.eclipse.equinox.common	同上



注意 Eclipse的插件JAR文件名由插件名称加上版本号后缀构成, 寻找插件时无须关心后缀。



JFace的查看器框架包括查看器、控件、数据模型和内容/标签提供者4部分, 前两者已经由SWT/JFace提供了, 所以使用查看器开发程序, 就是要编写后面两部分, 然后在主程序中将这4部分内容组合起来。使用查看器的一般步骤如下:

第一步: 定义元素类、设计数据模型。

第二步：编写内容提供者将数据模型中的元素集合提取出来；编写标签提供者将每一个元素转化成文本/图片以用于显示。

第三步：创建控件。

第四步：创建查看器对象，将内容/标签提供者安装到查看器上，并为查看器设定数据模型。

第五步：操作数据模型以在控件中显示希望的结果。

现在用列表查看器开发一个显示用户信息的程序。首先，定义了作为查看器元素的User类型，这是一个有两个属性(id和name)的数据类型，程序中用它来存储用户的信息，代码如下所示。

```
public class User {  
    private String id;  
    private String name;  
    //get和set方法  
    .....  
}
```



程序中所使用的数据模型ListModel维护了一个User对象的列表，可以通过ListModel的add和remove方法向这个列表中添加或删除User，当这些操作发生时，ListModel会向监听者发出“添加了一个User对象”或“删除了一个User对象”的通知消息。代码如下所示。

```
public class ListModel {  
    .....  
}
```

```
public void add(Object element) {
```

开发者可以通过add/remove方法修改数据模型的内容

```
    if (content.add(element))
```

```
        firePropertyChange(new PropertyChangeEvent(this,
```

```
        ADD_ELEMENT, null, element));
```

```
public void remove(Object element) {
```

```
    if (content.remove(element))
```

```
        firePropertyChange(new PropertyChangeEvent(this,
```

```
        REMOVE_ELEMENT, null, element));
```

在模型上添加或删除一个对象时，模型都会发出对应的通知消息

将数据模型的内容已经改变的消息通知给监听器

```
public void firePropertyChange(PropertyChangeEvent evt) { .....}
```

```
public void addPropertyChangeListener(PropertyChangeListener  
listener) { .....}
```

```
public void removePropertyChangeListener(PropertyChangeListener  
listener) { .....}
```

可以在这个数据模型上添加或移除对模型改变的监听器

开发者调用Viewer.setInput方法将数据模型传给查看器后，查看器会将模型传递给内容提供者，内容提供者则需要向数据模型上注册监听器。为列表查看器提供的内容提供者实现了IStructuredContentProvider接口，在实现内容提供者时，需要关注以下两个方法。

★inputChanged() —— 向数据模型注册监听器；

★getElements() —— 从数据模型中取出元素集合，提供给查看器。

另外，当内容提供者监听到模型的变化（添加或删除元素）时，要负责操作查看器以将这些变化反映出来。表7-3列出了列表查看器提供的常用操作方法（查看器共有的方法如refresh等不再列出），这些方法是查看器对控件的对应操作所做的包装，调用它们就可以改变控件中显示的内容。

表7-3 列表查看器的操作方法

方法名	方法用途
add(Object element)	向列表查看器中添加一个元素，等价于向List控件中添加一个item
add(Object[] elements)	向列表查看器中添加一组元素
remove(Object element)	从列表查看器中删除一个元素，等价于从List控件中删除一个item
remove(Object[] elements)	从列表查看器中删除一组元素

方法inputChanged和getElements代码如下所示（代码见光盘：\book.ch7.ListContentProvider.java）

```

public class ListContentProvider implements IStructuredContentProvider,
    PropertyChangeListener {

    private ListView viewer;

    private ListModel model;

    public Object[] getElements(Object inputElement) {
        return model.elements();
    }

    public void inputChanged(Viewer viewer, Object oldInput,
        Object newInput) {
        this.viewer = (ListView) viewer;

        if (oldInput instanceof ListModel)
            ((ListModel) oldInput).removePropertyChangeListener(this);

        if (newInput instanceof ListModel) {
            this.model = (ListModel) newInput;
            ((ListModel) newInput).addPropertyChangeListener(this);
        }

        public void propertyChange(PropertyChangeEvent evt) {
            if (ListModel.ADD_ELEMENT.equals(evt.getPropertyName()))
                viewer.add(evt.getNewValue());
            if (ListModel.REMOVE_ELEMENT.equals(evt.getPropertyName()))
                viewer.remove(evt.getNewValue());
        }
    }
}

```

开发者调用列表查看器的setInput方法时，列表查看器调用这个方法从新的模型中取得需要的元素数组

当setInput方法被调用时，列表查看器调用内容提供者的这个方法来自知模型已经改变

内容提供者停止监听旧模型，并开始监听新模型的变化情况

内容提供者根据收到的通知内容(模型上添加或删除了一个元素)，来相应地向查看器上添加或删除元素

当列表管理器通过内容提供者的getElements方法取得元素集合后，下一步需要将集合中的元素显示在控件中，这时查看器会调用标签提供者将元素翻译成文字和图片，列表查看器所对应的标签提供者——ILabelProvider提供了如下方法。

★getImage(Object element) —— 输入元素，返回代表元素的图片；

★getText(Object element) —— 输入元素，返回代表元素的文字。

其代码如下所示（代码见光盘：光盘：\book.ch7.Listviewer\ListLabelProvider.java）。

```
public class ListLabelProvider implements ILabelProvider {
```

```
    public Image getImage(Object element) {
        return null;
    }
```

为元素指定一个图标，这个图标会显示在该元素在List中所对应的item上

```
    public String getText(Object element) {
        if(element instanceof User)
            return ((User)element).getName();
        return element.toString();
    }
```

根据元素内容得到需要显示的文字。如果将这里改成User.getId，列表中将显示用户的id而不是名字

.....

```
}
```

这些部件全部完成后，最后一步就是将它们安装到查看器上，然后就可以通过操纵数据模型来改变控件的显示内容了。下面的代码演示了如何使用列表查看器显示用户信息（代码见光盘：\UsingListview.java）。

```
List list = new List(shell, SWT.BORDER);
ListViewer viewer = new ListViewer(list);
```

创建一个List控件，并创建一个基于它的查看器。这里也可以直接创建一个查看器，并由查看器来负责创建List控件

```
viewer.setContentProvider(new ListContentProvider());
viewer.setLabelProvider(new ListLabelProvider());
```

为查看器设置内容提供者和标签提供者

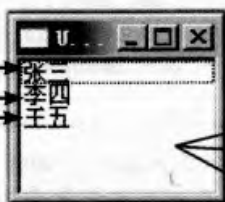
```
ListModel input = new ListModel();
viewer.setInput(input);
```

创建一个数据模型并将它输入到查看器

```
shell.open();
shell.layout();
```

在数据模型中添加数据元素，这时查看器中就会显示对应的内容。这是数据模型，查看器和内容/标签提供者共同作用的结果

```
input.add(new User("1", "张三"));
input.add(new User("2", "李四"));
input.add(new User("3", "王五"));
```



与标签提供者的实现相对应，这里显示的是用户的姓名



注意：

创建查看器时，一定要按照先设定内容/标签提供方，后设定数据模型
的顺序编写代码，否则查看器会报错。



使用查看器编写操作复杂控件的代码，每一个组件各自负责一部分功能，结构比较清晰，代码较容易维护和修改；而且可以通过更换或修改某一个或几个组件来方便地扩展已有的查看器的功能，应对需求变化的能力较强。在使用复杂控件时，应该坚持使用查看器。

7.1.3 Table控件

与List相似，Table通常用于显示列表，Table可以同时显示多列数据，而且支持显示图片。图7-5演示了如何使用Visual Editor创建一个Table。首先新建了一个Shell，选择需要的布局(示例选择FillLayout)，然后在SWT控件的面板中选择Table组件添加到Shell上。再选择Tree/Table Column组件，单击界面上的Table就可以为表格添加一栏(Column)。



图7-5 使用Visual Editor创建Table

选中Table，可以在属性视图中修改它的显示样式。BORDER样式为Table添加了一个粗线条的边框，而CHECK样式则会使Table的第一栏带有一个复选框。headerVisible和lineVisible两个属性分别控制着在Table中是否显示栏首和是否显示网格的线条。它们是可以动态改变的，可以使用setHeaderVisible和setLineVisible方法来控制这两个属性的值。

选中所添加的栏，在属性视图中可以修改它的text和image属性，它们决定着在栏首上面显示的文字和图片。width属性决定着栏的初始宽度；如果将resizable属性设置成true的话，用户可以用鼠标拖动来改变各栏的宽度。

Visual Editor不支持向Table中加入记录，需要手动编写代码实现，在Table中加入一行记录的代码如下所示。


```
table = new Table(sShell, SWT.NONE);
table.setHeaderVisible(true);
table.setLinesVisible(true);
TableColumn tableColumn1 = new TableColumn(table, SWT.NONE);
tableColumn1.setWidth(60);
tableColumn1.setText("Column 1");
tableColumn1.setImage(image1);
TableColumn tableColumn2 = new TableColumn(table, SWT.NONE);
tableColumn2.setWidth(60);
tableColumn2.setText("Column 2");
tableColumn2.setImage(image2);
```

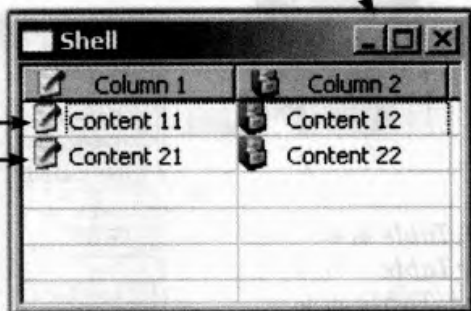
由Visual Editor
生成的代码，创
建了一个Table控
件，添加了两栏
并设定它们的宽
度和图片

```
TableItem item1 = new TableItem(table, SWT.NONE);
item1.setText(new String[] { "Content 11", "Content 12" });
item1.setImage(new Image[] { image1, image2 });
```

手动添加的代码，为表格添加
了两行内容

```
TableItem item2 = new TableItem(table, SWT.NONE);
item2.setText("Content 21");
item2.setText(1, "Content 22");
item2.setImage(image1);
item2.setImage(1, image2);
```

每一个TableItem对应
着表中的一行内容

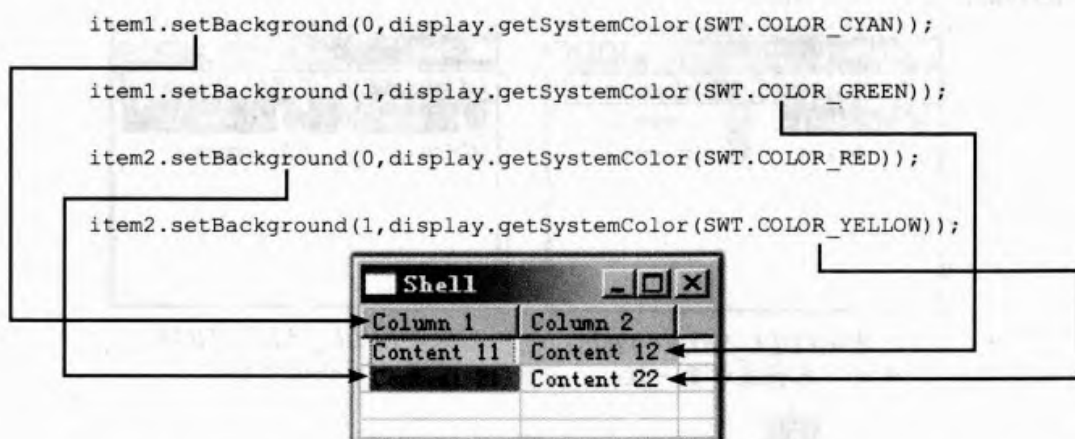


示例代码中使用了不同的方法为两个TableItem的各栏设置文字和图片。表7-4列出了可以用来设置文字的方法及说明（设置图片的情况类似）。

表7-4 为TableItem设置文字的方法

方法名	方法用途
setText(String[] texts)	使用String数组为TableItem设置文字，如果数组的长度超过表格的栏数，多余的内容将被忽略
setText(int column, String text)	为column所指定的栏位设置文字
setText(String text)	为第一栏设置文字，等同于setText(0, text)

除文字和图片外，包括前景色、背景色、字体等都是可以单独设定的。下面的代码以背景色为例演示了这一点。



如果想从Table中删除一个TableItem（等于从Table中删除一行记录），可以使用remove方法，表7-5列出了Table支持的多种用于删除TableItem的方法。

表7-5 删除TableItem的方法

方法名	方法用途
remove(int index)	删除第index行的TableItem
remove(int from, int to)	删除从from行到to行的TableItem
remove(int[] indices)	删除在indices数组中指定行的所有TableItem
removeAll()	删除所有TableItem

下面的代码从Table中删除了第4行（0代表第一行）的内容。

```
table.remove(3);
```

如果创建Table时使用了CHECK样式，在Table的第一列中会为每一行显示一个复选框，如图7-6所示，可以用TableItem.getChecked方法得知某一行的复选框是否被勾选，而setChecked方法则可以修改这个状态。使用TableItem.setGrayed方法可以使对应复选框的背景显示成灰色，但这并不影响选择动作。

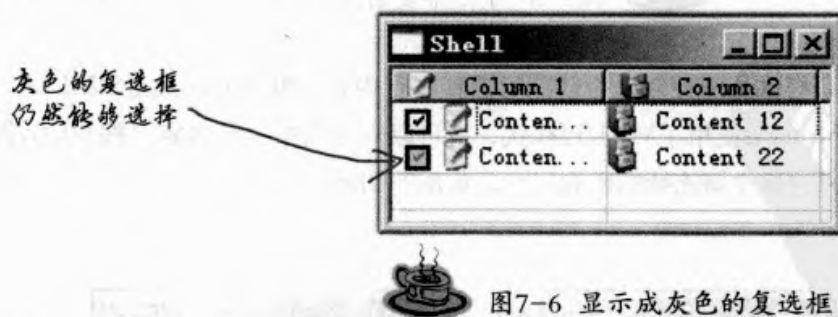
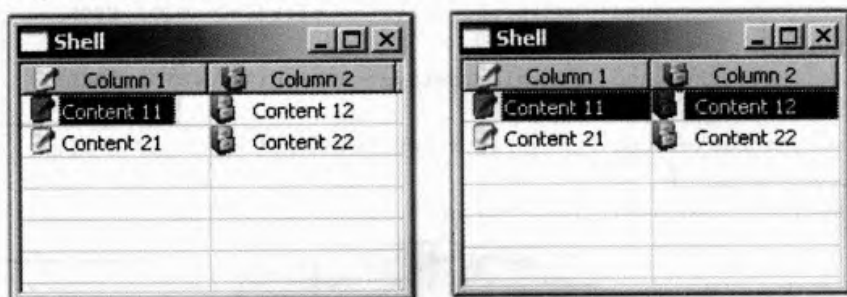


图7-6 显示成灰色的复选框

用户在Table中单击鼠标时，被选中的行会反白显示，FULL_SELECTION, HIDE_SELECTION和SIMPLE/MULTI这三组样式可以用来控制Table在被选择时的外观。

默认情况下，只有在第一列中单击的时候，才能选中某一行，这时被选中的行中也只有第一列的内容会反白，如果使用了FULL_SELECTION样式，单击任意一列都可以选中某一行，而且选中的行

中所有的列都会反白显示，如图7-7所示。



未使用FULL_SELECTION
样式，只能选中第一列

使用FULL_SELECTION
样式，可以选中整行



图7-7 为Table使用FULL_SELECTION样式

而HIDE_SELECTION样式则用于控制窗口失去焦点时是否仍然显示选中的行，如图7-8所示。

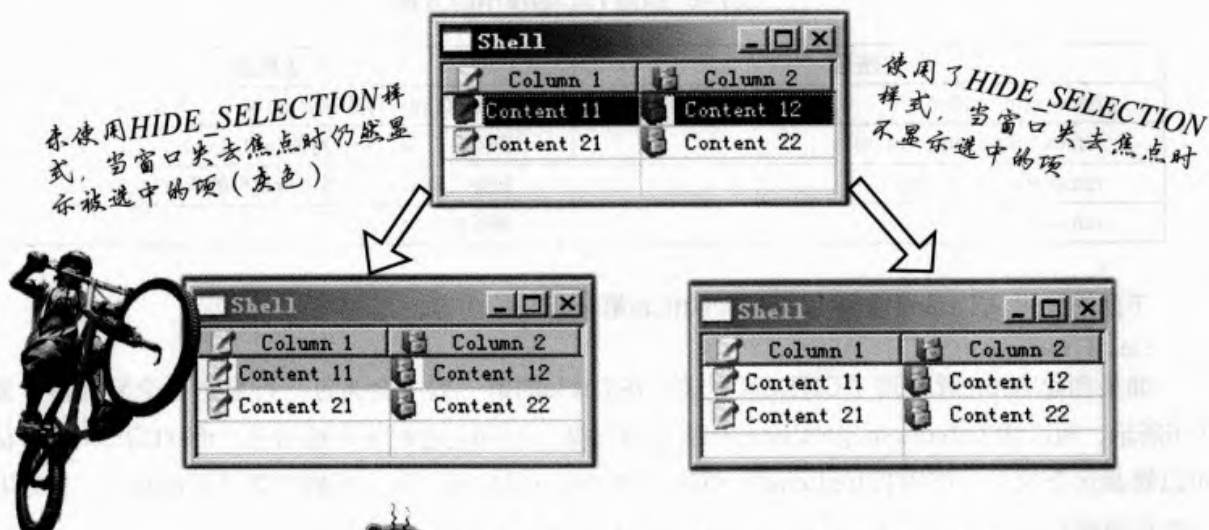
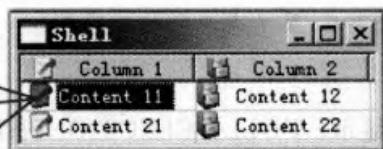


图7-8 为Table使用HIDE_SELECTION样式

SIMPLE和MULTI样式用于控制是否能同时选择多行内容，默认的样式是SIMPLE，这时用户只能选中Table中的一行；如果使用了MULTI样式，就可以同时选中多行内容（按住Ctrl键单击多行或按住Shift键单击两行来选择中间连续的所有行），如图7-9所示。

使用SINGLE
样式，只能选
中一行内容



使用MULTI
样式，可以同时
选中多行内容

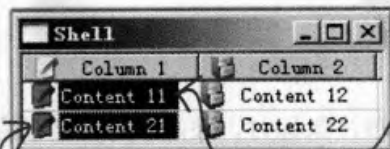


图7-9 Table控件的选择示例

Table提供了如下两个方法用于取得包含在其中的TableItem。

★getItem(int index) —— 返回在Table中序号为index的TableItem；

★getItem(Point point) —— 返回所占的像素区域包含参数点的TableItem。相当于用户用鼠标单击这一点时，所选中的TableItem。

当Table被选中时，会触发一个Selection事件，这时可以用getSelectionIndices方法取得被选中的TableItem的索引，然后用getItem方法得到对应的TableItem并加以处理。

7.1.4 JFace 表格查看器(Table Viewer)

查看器框架将不同控件的区别封装在了不同的查看器内部，而在外部保留了相似的接口（如设定数据模型、设定内容、标签提供者等），因此使用各种查看器的方式基本一致。

下面的代码演示了如何使用表格查看器同时显示用户的id和姓名（代码见光盘：book.ch7. TableViewer\UsingTableViewer.java）。

```
Table table = new Table(shell, SWT.BORDER);
TableColumn column1 = new TableColumn(table, SWT.NONE);
column1.setWidth(100);
TableColumn column2 = new TableColumn(table, SWT.NONE);
column2.setWidth(100);
table.setHeaderVisible(true);
table.setLinesVisible(true);
```

创建Table控件，设定其样式，并向其中添加两个TableColumn

```
TableViewer viewer = new TableViewer(table);
viewer.setContentProvider(new TableContentProvider());
viewer.setLabelProvider(new TableLabelProvider());
```

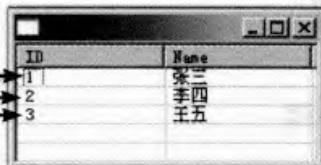
创建表格查看器，安装内容提供者和标签提供者

```
ListModel input = new ListModel();
viewer.setInput(input);
```

创建数据模型并将它输入到查看器

```
shell.open();
shell.layout();
```

```
input.add(new User("1", "张三"));
input.add(new User("2", "李四"));
input.add(new User("3", "王五"));
```



ID	Name
1	张三
2	李四
3	王五



由于列表和表格所需的数据结构都是对象数组，程序中仍然使用了第7.1.2节中用过的ListModel类作为模型。表格查看器的内容提供者是TableContentProvider，它同样继承自IStructuredContentProvider，除了将viewer属性的类型从ListViewer换成了TableViewer外，TableContentProvider和

ListContentProvider的代码也完全相同。表7-6列出了在内容提供者中可能使用到,且表格查看器所支持的操作。

表7-6 表格查看器支持的操作

方法名	方法用途
add(Object element)	添加一个元素
add(Object[] elements)	添加一组元素
remove(Object element)	删除一个元素
remove(Object[] elements)	删除一组元素
replace(Object element, int index)	使用给定的元素替代序号为index的元素
clear(int index)	删除序号为index的元素

TableLabelProvider的变化就比较大了,它继承自ITableLabelProvider。Table是要分栏显示内容的,所以与列表查看器所使用的ILabelProvider相比,在它的方法的参数中多了“行号”(ColumnIndex)一项。

★getColumnImage(Object element, int columnIndex) —— 从输入元素中取得代表第columnIndex列的图片;

★getColumnText(Object element, int columnIndex) —— 从输入元素中取得代表第columnIndex列的文字。

其代码如下所示。

```
public Image getColumnImage(Object element, int columnIndex) {
    return null;
}

public String getColumnText(Object element, int columnIndex) {
    if (element instanceof User) {
        User user = (User) element;
        switch (columnIndex) {
            case 0:
                return user.getId();
            case 1:
                return user.getName();
        }
    }
    return null;
}
```

从User元素中取出用于显示的文字,第一栏显示用户ID,第二栏显示姓名



7.1.5 Tree控件

使用Visual Editor创建一个Tree的步骤与创建Table基本相似。Tree控件可以使用的各种样式也与Table相差无几。也可以使用Tree/Table Column组件向Tree中加入几栏并修改它们的文字/图片。下面代码是由VE生成,用于显示Tree控件。

```
tree = new Tree(sShell, SWT.CHECK);
```

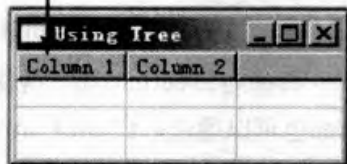
```
tree.setHeaderVisible(true);
tree.setLinesVisible(true);
```

向Tree中添加一栏时，使用的是TreeColumn而不是TableColumn，但两者所支持的方法基本一致

```
TreeColumn treeColumn = new TreeColumn(tree, SWT.NONE);
treeColumn.setWidth(60);
treeColumn.setText("Column 1");
```

为树创建一个栏，并设置其宽度和名称

```
TreeColumn treeColumn1 = new TreeColumn(tree, SWT.NONE);
treeColumn1.setWidth(60);
treeColumn1.setText("Column 2");
```



Tree中显示的树状节点是由TreeItem代表的，在VE生成的代码中添加如下内容。

与TableItem不同，TreeItem即可以Tree为参数创建，又可以其他的TreeItem为参数创建。

```
TreeItem root1 = new TreeItem(tree, SWT.NONE);
root1.setText(new String[] { "Root 1", "Root 1 Content" });
```

```
TreeItem root2 = new TreeItem(tree, SWT.NONE);
root2.setText(new String[] { "Root 2", "Root 2 Content" });
```

```
TreeItem child1 = new TreeItem(root1, SWT.NONE);
child1.setText("Child 1");
```

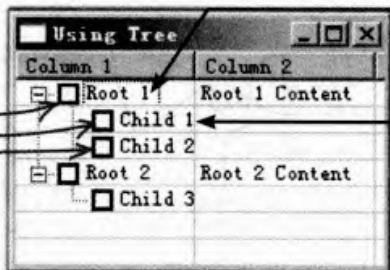
```
TreeItem child2 = new TreeItem(root1, SWT.NONE);
child2.setText("Child 2");
```

```
TreeItem child3 = new TreeItem(root2, SWT.NONE);
child3.setText("Child 3");
```

设置item每一栏的文字的方法与使用TableItem类似。Tree控件也可以在item中显示图片

直接在Tree上创建的TreeItem会显示成树的根节点

为Tree使用了CHECK样式，在每一个Item前面会显示复选框



在某一个item上创建的item会显示成这个item的子节点

TreeItem不仅可以用Tree为父资源创建，也可以用其他节点为父资源创建出来。在Tree中，各个节点就是按照这种资源从属关系显示成树状的结构。

※ 注意 : ※

在旧的SWT版本中，提到“Tree”的时候，所指的是只有一栏的Tree控件，而像上面这种拥有多栏、很像表格的Tree控件，是由“TableTree”控件代表的，从Eclipse 3.1开始，为了避免混淆，SWT将两者的功能合并在了Tree控件中。SWT中的TableTree控件已经被置为“Deprecated”，而JFace中虽然还有对应的查看器TableTreeViewer，但已经没必要使用，使用TreeViewer就可以满足全部需求了。



Tree与Table有很多类似的地方，如下所示。

★对Tree的选择会触发一个Selection事件，监听器收到这个事件后可以使用Tree.getSelection方法返回所有被选中的TreeItem。如果使用MULTI样式创建Tree，就可以同时选中多个节点。

★Tree也可以使用CHECK样式，TreeItem提供了get,setChecked方法来取得并控制每一个节点的复选框状态。

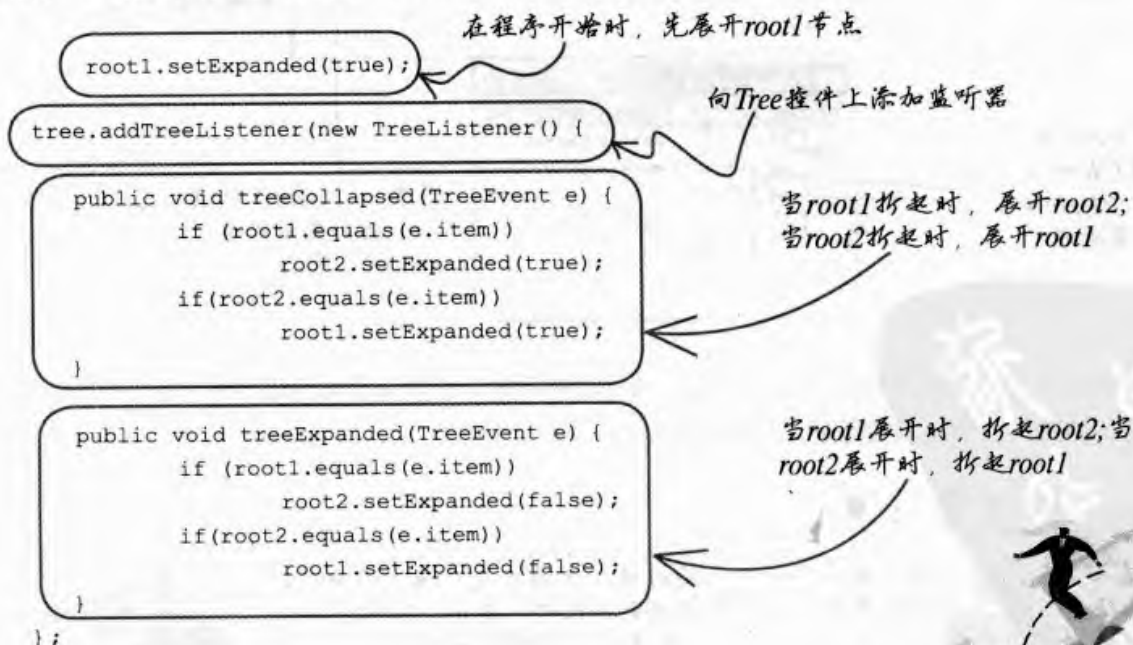
★可以为某个TreeItem单独定义某一栏的颜色、字体、前景、背景等属性。

Tree与TreeItem都支持使用getItem(int index)方法通过序号查找资源属于自己的节点，另外Tree也可以通过getItem(Point point)方法查找到占有point的节点。调用Tree.showItem(TreeItem item)方法可以将所指定的TreeItem显示在窗口中(这个子节点的所有父节点都会被展开，如果它不处在窗口区域内，Tree会移动滚动条保证这个节点在窗口内可见)。

Tree没有提供remove方法来删除一个节点，这一点与Table有所不同。如果需要从Tree上删除一个节点，可以从Tree开始逐层向下找到需要删除的TreeItem，然后调用dispose()方法来释放这个TreeItem以删除它代表的节点，当一个节点被删除时，它的所有子节点也将同时被删除。另外，Tree与TreeItem都提供了removeAll方法来删除所有子节点。

拥有子节点的节点在左侧会显示一个加号，单击它可以展开或折起其中所包含的结构，调用TreeItem.setExpanded(Boolean show)方法也可以达到同样的效果。

对Tree中的节点作展开/折起操作后，会触发Tree的Expand/Collapse事件。可以用TreeListener来监听它们，事件中包含了被展开/折起的TreeItem信息。下面的程序演示了如何使用这些事件。在前面的示例中，已经创建了两个根节点root1和root2。现在将通过监听这两个事件使得两个节点中有且仅有一个被展开(代码见光盘：\book.ch7.tree.UsingTree.java)。



当root1被用户展开时，程序折起root2；当root1被用户折起时，程序则展开root2；对root2也是同样处理。这样就实现了两个节点中，有且仅有一个节点被展开。

※ 注意 : ※

*Tree*的Expand事件和Collapse事件都是在展开或折起动作发生之后才发送出去的, 因此无法试图通过将事件类TreeEvent的doit属性设置为false来取消这个动作。另外, 调用TreeItem.setExpanded方法来展开或折起节点不会触发这个事件。



7.1.6 JFace树查看器 (Tree Viewer)

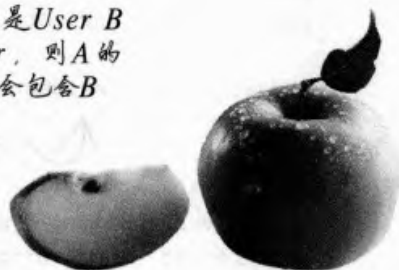
本节将使用树查看器在Tree控件中显示一个上下级的人事关系。

树查看器适合于显示层级化的数据模型, 因此首先要设计一个支持这种结构的数据类型。将之前使用的User类型改变为如下所示 (代码见光盘: \book.ch7.tree.User.java)。

```
public class User {  
    private String id;  
    private String name;  
    .....  
  
    private User manager;  
    public User getManager() {  
        return manager;  
    }  
    public void setManager(User manager) {  
        this.manager = manager;  
    }  
  
    private Vector underlings;  
    public Vector getUnderlings() {  
        return underlings;  
    }  
}
```

为User引入了manager属性, 如果一个User对象的manager属性为null, 则认为他是最高级别的人员。用Tree控件来显示User之间的这种层级关系

如果User A是User B的Manager, 则A的underlings中会包含B



现在设计一个支持树状结构的数据模型UserStructure, 这个模型维护着一个User对象作为树状结构的根节点, 如果用户需要访问模型中的元素, 需要指定从根节点到它的“路径”来取得该元素。路径是由一个int数组指定, 它由其父节点的路径加上该节点在父节点中的序号构成, 根元素的路径为空数组。图7-10演示了如何确定一个节点的路径。



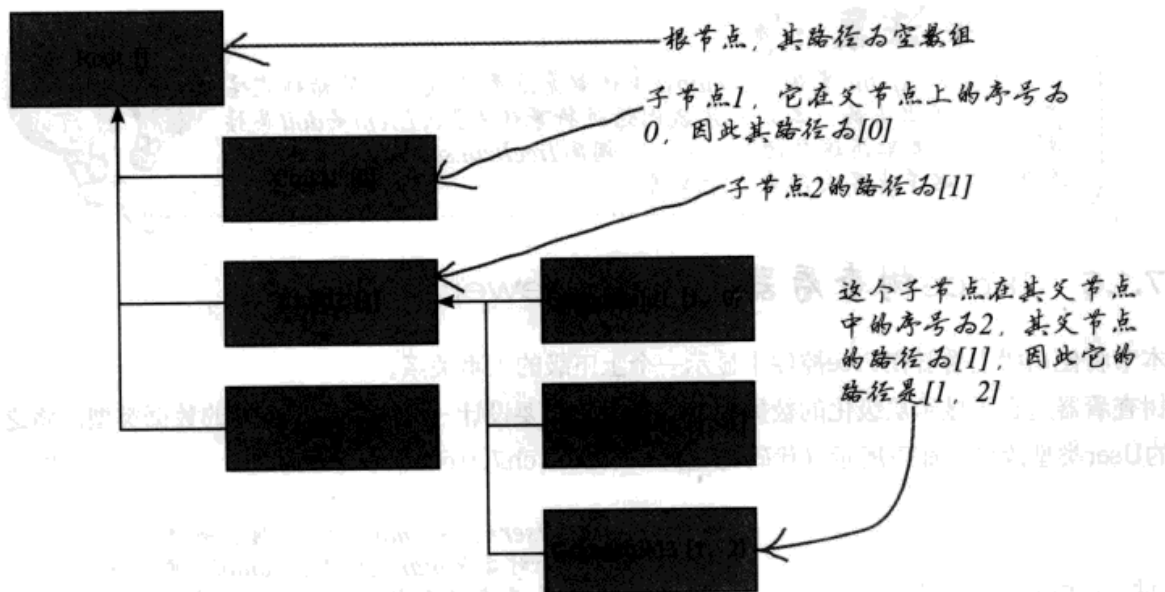


图7-10 在树状结构中确定一个节点的路径

在这个模型中, 开发者可以指定一个路径并向路径代表的节点上添加一个元素, 也可以通过路径指定删除一个元素, 这些操作都会使数据模型发送一个变化消息到监听器。

数据模型的代码如下所示 (代码见光盘: \book.ch7.tree.UserStructure.java)。

```
public class UserStructure {
    public static final String ADD_USER = "addUser";
    public static final String REMOVE_USER = "removeUser";
```

```
    private User president;
```

级别最高的用户,
树状结构的根节点

```
    .....
```

```
    public void add(int[] parentPath, User newUser) {
        User parent = findUser(parentPath);
        if (parent != null
            && !parent.getUnderlings().contains(newUser)) {
            if (parent.getUnderlings().add(newUser)) {
                newUser.setManager(parent);
```

在parentPath所指定的User
节点上添加一个作为子节点
的User

```
                firePropertyChange(new PropertyChangeEvent(this,
                    ADD_USER, null,
                    new Object[] { parentPath, newUser }));
```

在结构中添加用户时, 模型会向监听器
发出消息, 通知它们所添加的用户及添
加的位置

```
public void remove(int[] path) {
    User theUser = findUser(path);
    if (theUser != null &&
        theUser.getManager().getUnderlings().remove(theUser)) {
        theUser.setManager(null);
        firePropertyChange(new PropertyChangeEvent(this,
            REMOVE_USER, null, theUser));
    }
}
.....
}
```

删除数据模型的节点时，同样通知监听器

在数据模型中删除由路径path所指定的节点

树查看器的内容提供者需要实现ITreeContentProvider接口，它的结构比之前使用的IStructuredContentProvider要复杂许多，同时为了监听模型变化的事件，它还实现了PropertyChangeListener接口。在ITreeContentProvider中，主要需要关心以下几个方法。

★inputChanged —— 与之前使用的内容提供者相同，在这个方法中，需要将内容提供者注册到数据模型上，开始监听其变化，并当模型变化时通知查看器更新控件。

★getElements —— 在这个方法中，内容提供者要从数据模型中取出作为树的根节点的所有元素。

★getParent(Object element),getChildren(Object element)和hasChildren(Object element) —— 当树查看器需要取得一个元素的上下文信息——父节点和子节点时，会调用这些方法，内容提供者需要根据数据模型解析并返回这些信息。树查看器在调用getElements得到根节点信息后，就是调用了这些方法来遍历整个树状结构。

表7-7列出了在内容提供者中可以用来操作树查看器的方法。

表7-7 操作树查看器的方法

方法名	方法用途
add(Object parent, Object child)	向parent代表的元素上添加一个子元素
add(Object parent, Object[] children)	向parent代表的元素上添加多个子元素
remove(Object element)	删除element代表的元素
remove(Object[] elements)	删除多个元素
remove(Object parent, Object[] children)	删除parent元素下由children指定的子元素

内容提供者的代码如下所示（代码见光盘：\book.ch7.tree.UserTreeContentProvider.java）。




```
public Object[] getElements(Object inputElement) {
    if (inputElement instanceof UserStructure)
        return new Object[]
            { ((UserStructure) inputElement).getPresident() };
    return new Object[0];
}
```

从数据模型中取得根节点——*president*作为根节点返回

```
public Object[] getChildren(Object parentElement) {
    return ((User) parentElement).getUnderlings().toArray();
}

public Object getParent(Object element) {
    return ((User) element).getManager();
}

public boolean hasChildren(Object element) {
    Vector underlings = ((User) element).getUnderlings();
    return !(underlings == null || underlings.size() == 0);
}
```

从参数节点中解析并返回上下文信息

```
public void propertyChange(PropertyChangeEvent evt) {
    if (UserStructure.ADD_USER.equals(evt.getPropertyName())) {
        Object[] values = (Object[])evt.getNewValue();
        viewer.add(values[0], values[1]);
    }
    if (UserStructure.REMOVE_USER.equals(evt.getPropertyName())) {
        viewer.remove(evt.getNewValue());
    }
}
```

当模型的内容变化时，内容提供者负责相应地改变查看器的内容(使用查看器的add/remove方法添加或删除元素)

树查看器封装了对Tree控件的操作

```
public void inputChanged(Viewer viewer, Object oldInput,
    Object newInput) {.....}
```

与前面的查看器相似，在这个方法中，内容提供者开始监听作为输入的模型变化情况

树查看器可以使用和表格查看器一样的ITableLabelProvider，这里直接使用上一节中用过的TableLabelProvider。下面这段代码将各个部件组合在一起，创建树查看器并显示内容，运行效果如图7-11所示。

```
TreeViewer treeViewer = new TreeViewer(tree);
treeViewer.setContentProvider(new UserTreeContentProvider());
treeViewer.setLabelProvider(new TableLabelProvider());
```

创建树查看器，并为其设定内容提供者和标签提供者

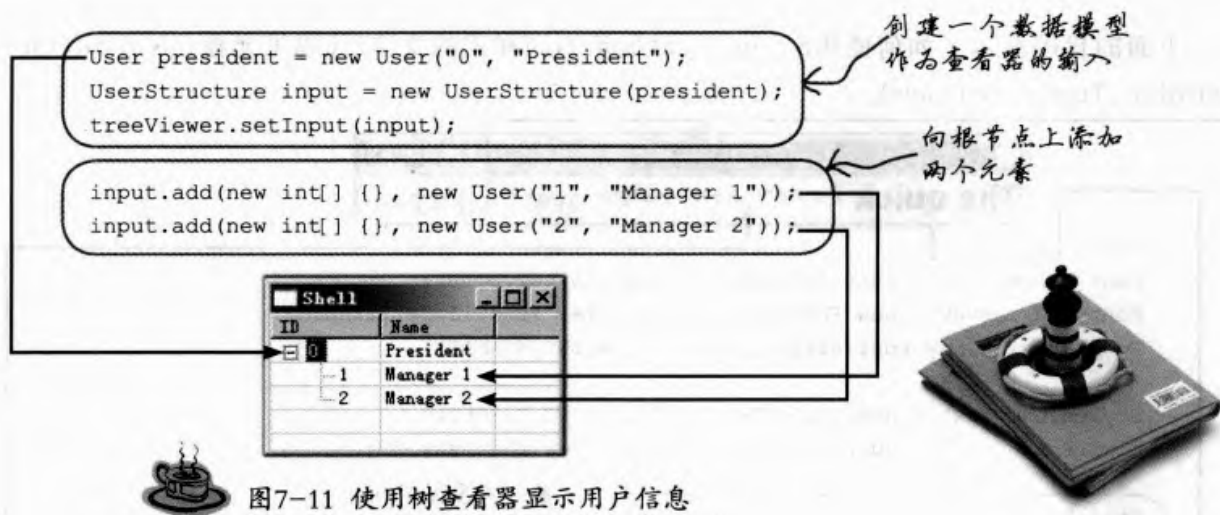


图7-11 使用树查看器显示用户信息

7.2 文本编辑器

使用Text控件，可以显示基本的文本内容，但是Text中的文字其字体、前景色、背景色等属性都是相同的，如果需要为不同的文字使用不同的属性，Text控件就无能为力了。这时，可以使用StyledText控件，StyledText可以显示多行文字，其中每一部分的文字都可以单独指定各种显示属性。如图7-12所示是一个用StyledText控件显示文字的例子。

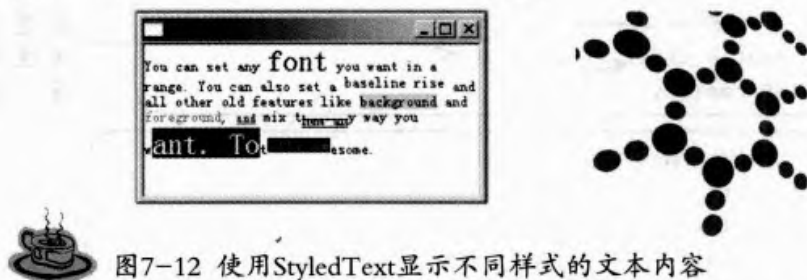


图7-12 使用StyledText显示不同样式的文本内容

在StyledText中为某一部分文本添加特殊显示效果时，首先要通过指定开始位置和长度来确定这一段文本，然后创建一个StyleRange对象。StyleRange类型中包含了所有特殊显示属性，设置好所需要的属性后，将StyleRange对象应用到所确定的文本，就可以使文本按照需要的格式显示出来了。图7-13演示了这一过程。



图7-13 使用StyledText显示文字的步骤

下面的代码演示了如何使用StyledText显示不同样式的文字(代码见光盘:\book.ch7.texteditor.TextEditor1.java)。

```

.....
Font tahoma = new Font(display, "Tahoma", 16, SWT.BOLD);
Font courierNew = new Font(display, "Courier New", 12, SWT.ITALIC);
Font arial = new Font(display, "Arial", 14, SWT.NORMAL);

StyledText text = new StyledText(shell, SWT.BORDER);
text.setText("The quick brown fox jumps over the lazy dog.");

StyleRange style1 = new StyleRange();
style1.start = 0;
style1.length = 9;
style1.font = tahoma;
text.setStyleRange(style1);

StyleRange style2 = new StyleRange();
style2.start = 10;
style2.length = 15;
style2.font = courierNew;
style2.foreground = display.getSystemColor(SWT.COLOR_BLUE);
style2.rise = 5;
style2.strikeout = true;
text.setStyleRange(style2);

StyleRange style3 = new StyleRange();
style3.start = 26;
style3.length = 14;
style3.font = arial;
style3.underline = true;
style3.foreground = display.getSystemColor(SWT.COLOR_RED);
style3.background = display.getSystemColor(SWT.COLOR_YELLOW);
text.setStyleRange(style3);
.....

```

第一个StyleRange, 从第0个字符开始, 长度为9。设定其字体为16号的Tahoma

第二个StyleRange从第10个字符开始, 长度为15。设置其字体为12号的Courier New, 斜体, 前景色为蓝色

设置这段文字在正常位置上向上移5像素, 加删除线

这个StyleRange从第26个字符开始, 长度为14。字体为14号Arial。带有下画线。前景色为红色, 背景色为黄色

表7-8列出了StyleRange中常用的设置选项。

表7-8 StyleRange的设置

设置项	含义
Font font	字体
Color foreground	文字前景色
Color background	文字背景色
boolean underline	是否有下画线
boolean strikeout	是否有删除线
int rise	文字上升(负值代表下降)

另外, 在StyledText中还可以以行为单位调整文字的位置(左、右、居中对齐等)。这里的“行”指的是由回车(\n)所造成的分隔, 也可以称为“段落”。图7-14演示了如何控制各行文字的样式(代

码见光盘：\book.ch7.texteditor.TextEditor2.java)。

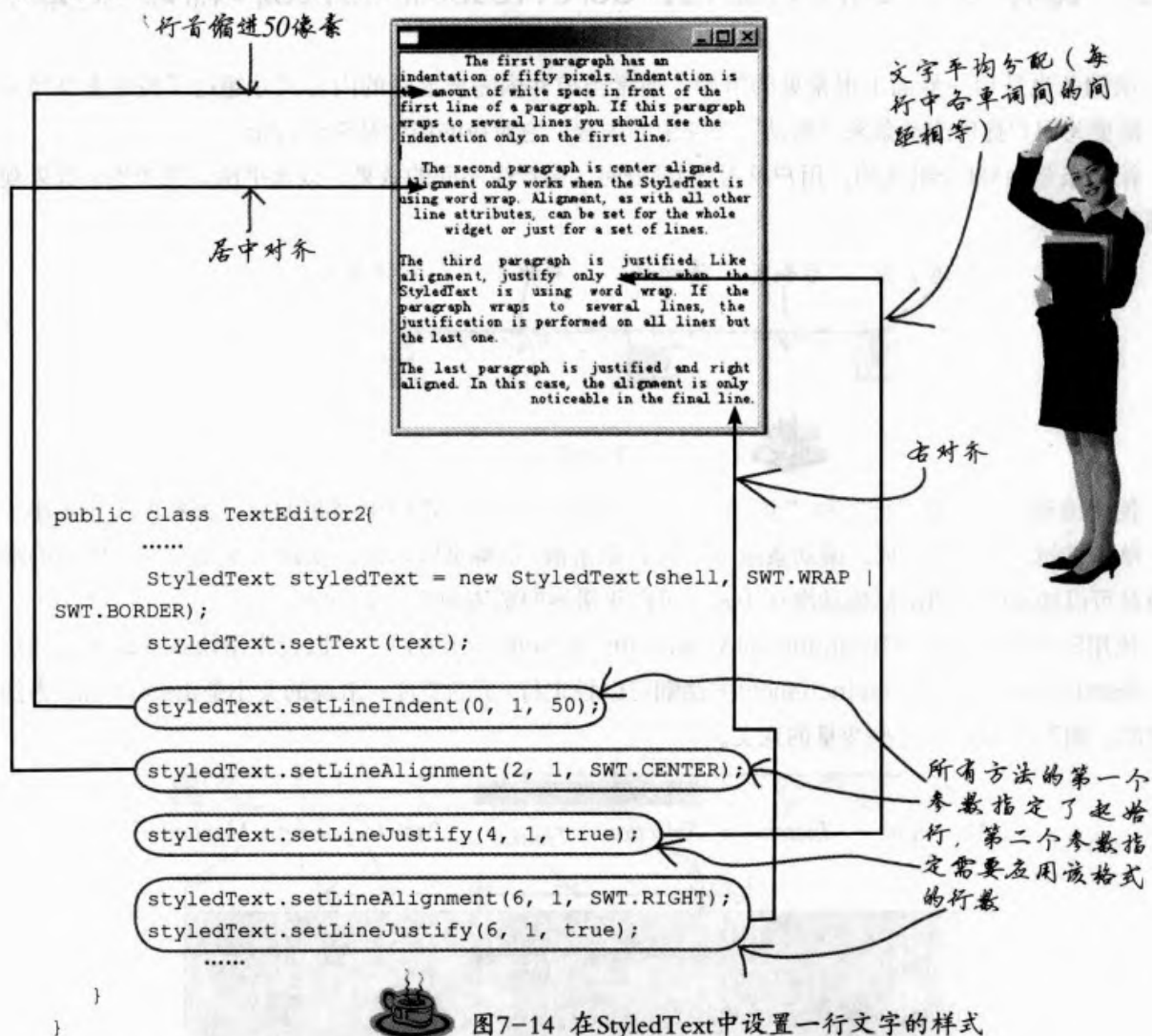


图7-14 在StyledText中设置一行文字的样式

表7-9列出了可用于设置段落格式的常用方法。

表7-9 用于设置段落格式的方法

方法名	方法用途
setLineAlignment(int fromLine, int lineCount, int alignment)	设置段落对齐方式, alignment可取值SWT. LEFT, SWT. RIGHT或SWT. CENTER
setLineBackground(int fromLine, int lineCount, Color bgcolor)	设置段落背景色
setLineIndent(int fromLine, int lineCount, int indent)	设置段落的首行缩进(以像素表示)
setLineJustify(int fromLine, int lineCount, boolean justify)	将段落设置成单词间距相等

7.3 滚动条、Scrollable、ScrolledComposite和滑动条

滚动条也是用户界面上很常见的组件，通常当控件内需要显示的内容尺寸超过了控件本身尺寸时，需要为用户提供滚动条来“滚动”控件中的内容，它对应的控件是ScrollBar。

滚动条是由5部分组成的，用户单击这5部分时，会产生不同的效果。以水平滚动条为例，其外观如图7-15所示。



图7-15 滚动条示例

使用滚动条时，有“行”和“页”的概念。通常一页的大小等于控件的尺寸，而行的大小要小于页。单击前进/后退箭头时，滚动条滚动一行；单击前/后翻页区域时，滚动条滚动一页。中间的滑动块是可以拖动的，用鼠标拖动滑动块时，可以平滑地向前或向后滚动内容。

使用ScrollBar的setMinimum/setMaximum/setSelection方法，可以设定滑块在滚动条上的位置；而setIncrement/setPageIncrement方法则可以设定行/页的宽度；滑块的大小是由setThumb方法指定的。图7-16显示了这些变量的意义。

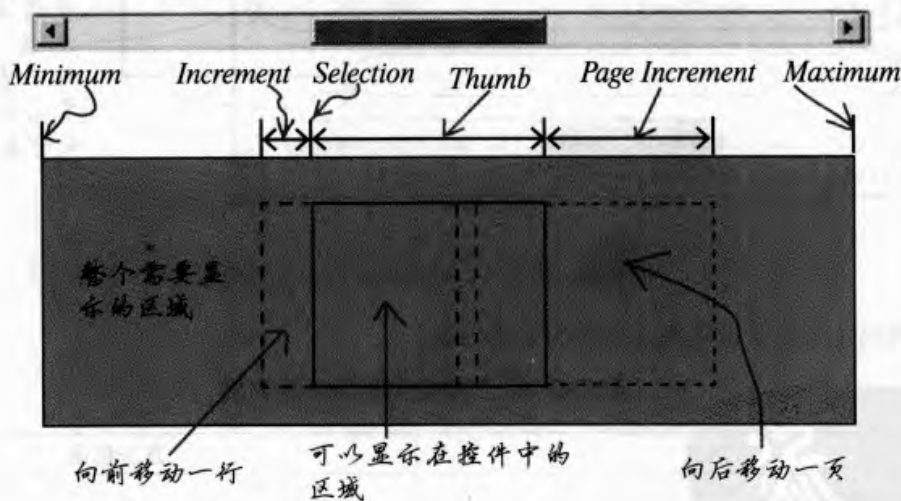


图7-16 滚动一行和滚动一页

SWT将显示滚动条的功能包装在了抽象类Scrollable中，所有继承Scrollable的控件类都可以通过使用H_SCROLL和V_SCROLL以显示标准的水平滚动条(处于控件下方)和竖直滚动条(处于控件右方)。可以使用Scrollable.getHorizontalBar和getVerticalBar方法来得到对应的滚动条对象。Composite,List和Text等控件都是Scrollable的子类。

需要注意，Scrollable所提供的功能只是显示滚动条，而不提供“滚动”其内部控件的功能。开发者需要自己监听滚动条的选择事件并相应地改变显示在控件中的内容，为了减轻开发者的负担，SWT提供了ScrolledComposite类并将这些烦琐的工作包装在了它内部，直接使用这个控件就可以方

便地为程序添加滚动条了。

下面的代码演示了如何使用ScrolledComposite在窗口中显示一张图片，当窗口尺寸小于图片时，会显示一个滚动条出来（代码见光盘：\book.ch7.scroll.UsingScrolledComposite.java）。运行效果如图7-17所示。

```
Shell shell = new Shell(display, SWT.SHELL_TRIM);
shell.setSize(300, 200);
shell.setLayout(new FillLayout());
```

创建用于在窗口中
显示的Image对象

```
Image image = new Image(display, UsingScrolledComposite.class
    .getResourceAsStream("dddgs04.jpg"));
```

```
ScrolledComposite sc = new ScrolledComposite(shell,
    SWT.HORIZONTAL|SWT.VERTICAL);
```

创建一个ScrolledComposite
对象，设定其样式为显示
水平和竖直的双滚动条

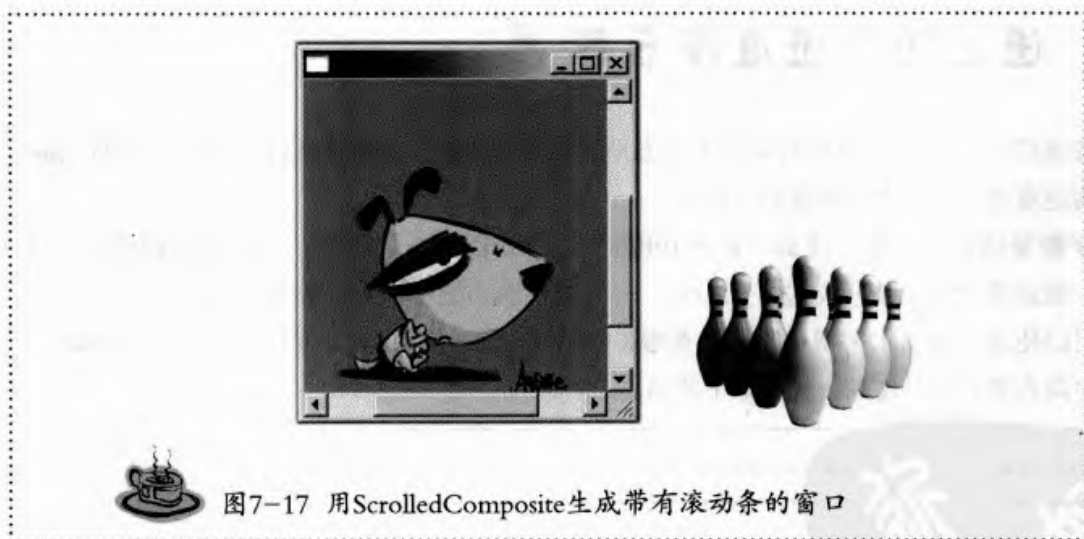
```
Label label = new Label(sc, SWT.NONE);
label.setImage(image);
label.pack();
```

将图片设置到Label上，并调用
pack方法，请求Label按照图片的
尺寸设置自己的尺寸

```
sc.setContent(label);
```

调用setContent方法将Label绑定到ScrolledComposite
上，这样ScrolledComposite就可以比较Label的尺寸
和自己的尺寸来决定是否显示滚动条

```
shell.open();
```



在某些图形平台上，滚动条不是一个单独的控件，而是某个控件的一部分。因此SWT也没有将滚动条设计成一个单独的控件，如果需要单独使用滚动条，可以考虑用滑动条替代。

滑动条(Slider)的外形、功能和使用方法都与滚动条完全类似，唯一的区别在于Slider是一个独立的控件，用户无法单独创建一个滚动条，但是可以创建一个Slider并随意安排它的位置。可以使用样式SWT.HORIZONTAL/SWT.VERTICAL来选择创建一个水平滑动条(默认)或竖直滑动条。当用户单击Slider时，同样会触发一个Selection事件。下面的代码演示了如何创建一个Slider并监听它的选择事件(代码见光盘：\book.ch7.scroll.UsingSlider.java)。运行效果如图7-18所示。


```
Slider slider = new Slider(shell, SWT.HORIZONTAL);
slider.setSelection(0);
```

创建一个水平滚动条，设置其位置为最左边(对应值为0)

```
final Text text = new Text(shell, SWT.BORDER);
text.setText(String.valueOf(0));
text.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, false, false));
```

```
slider.addSelectionListener(new SelectionAdapter() {
```

```
    public void widgetSelected(SelectionEvent e) {
        Slider slider = (Slider)e.widget;
        text.setText(String.valueOf(slider.getSelection()));
    }
};
```

当Slider被选择时，从其中取得新的选择值并在text控件中显示出来

在文本框中显示滑动条的当前值

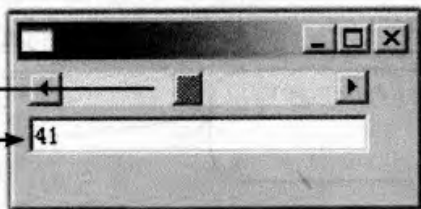


图7-18 使用Slider

7.4 进度条与进度指示器

进度条(ProgressBar)控件可以用来动态地显示工作进度，进度条无法接受用户键盘或鼠标的输入，它的进度改变必须用代码在后台控制。

对于数量确定的工作，比如“循环100次”，“处理20条记录”，可以使用确定样式(默认)的进度条，表示进度的色块从左边开始(0%)，一直增长到右边(100%)。如果使用SWT.VERTICAL样式，也可以使进度条由水平增长改为竖直增长(由下向上)。使用SWT.NONE样式可以创建一个普通的、由左向右增长的进度条，代码如下所示。

```
ProgressBar progressBar = new ProgressBar(this, SWT.NONE);
```

与滚动条类似，使用进度条的setMinimum, setMaximum和setSelection可以控制进度条的显示。前两个方法分别用于指定进度条的最小值和最大值，第三个方法用来指定进度条的当前进度。当前进度等于最小值时，进度条显示为0%，进度等于最大值时，进度条显示成100%。即有公式：进度(%)= 100% × (Selection - Minimum) / (Maximum - Minimum)。以用进度条显示循环100次的工作为例，可以将minimum设为0，maximum设为100。selection初始值为0，循环一次就将selection的值加1。

对于不能确定工作量的工作，如等待网络服务器响应的情况，可以使用INDETERMINATE样式的进度条，如图7-19所示。使用这种样式时进度条不显示成稳定增长的样子，而是不断地盘旋。进度条处于这种样式时，设置它的最小值，最大值以及当前进度等都是无效的。



图7-1 INDETERMINATE样式的进度条

默认情况下，进度条显示成一格一格的样子，使用SMOOTH样式可以使进度条显示成连续的一条色块，如图7-20所示。



图7-20 SMOOTH样式的进度条

进度指示器(ProgressIndicator)是JFace以ProgressBar为基础编写的控件，它克服了进度条控件的一些缺陷，如创建以后就不能改变状态等，并重新包装了接口，使用起来更加方便直观。进度指示器继承自Composite类，可以像使用普通控件一样创建它。

进度指示器创建了两个ProgressBar控件，一个是默认样式（确定的进度条），一个则是INDETERMINATE样式的，并用StackLayout将它们叠放了起来。这样在进度指示器运行时，可以通过切换StackLayout的topControl来在两个ProgressBar中切换。因此无论是确定的任务或是不确定的任务，进度指示器都能够处理。

进度指示器提供了以下几个方法来控制进度条的状态，如表7-10所示。

表7-10 控制进度指示器的方法

方法名	方法用途
beginTask()	显示确定样式的进度条
beginAnimatedTask()	显示不确定样式的进度条
worked(double work)	将已经完成的工作进度加到确定样式的进度条上
done()	结束进度指示工作，隐藏进度条
sendRemainingWork()	使确定样式的进度条进度变为100%

图7-21演示了使用进度指示器分别显示确定性的任务和不确定性任务的调用流程。当进度指示器处于不显示状态时，开发者可以调用beginTask方法以进入显示确定性任务的流程，或者调用beginAnimatedTask以进入显示不确定性任务的流程。在前一个流程中，开发者需要反复调用worked方法或调用sendRemainingWork方法以使确定性的进度条达到100%处，然后调用done方法以完成整个流程，使进度指示器重新回到不显示的状态；而在后一个流程中，开发者随时可以调用done方法来结束不确定性进度条的显示并回到初始状态。当回到初始状态后，又可以选择一个流程并重新进入。



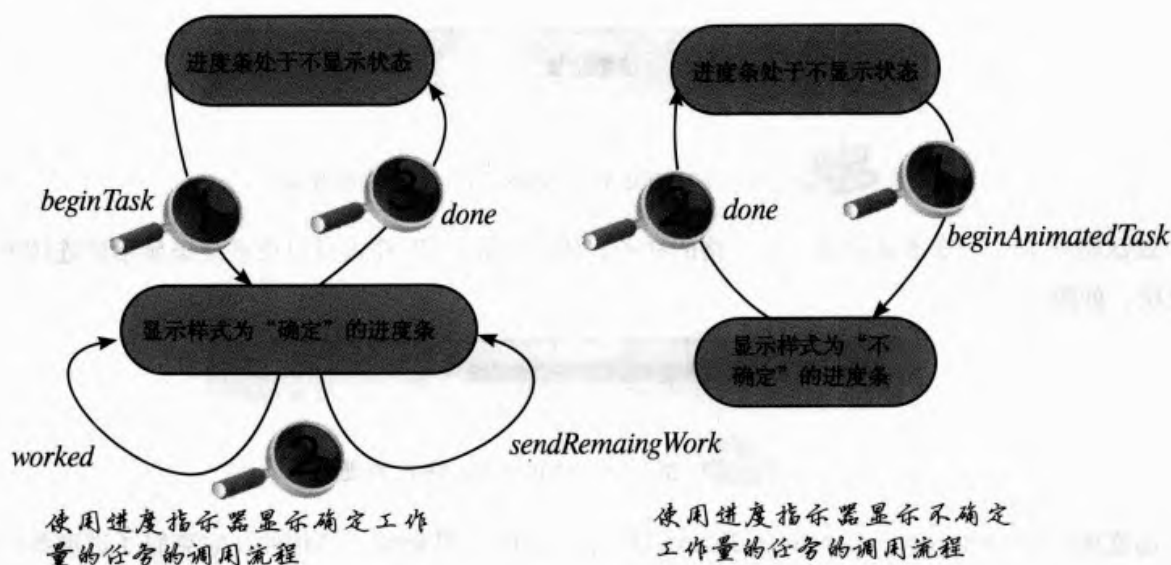


图7-21 使用进度指示器的两种流程

7.5 浏览器与OLE

在SWT中使用浏览器是非常简单的一件事情。以下几行代码在Shell中创建了一个浏览器组件，并打开www.eclipse.org网站。

```
Browser browser = new Browser(shell, SWT.NONE);
browser.setUrl("http://www.eclipse.org");
```

在Windows系统下的执行效果如图7-22所示。

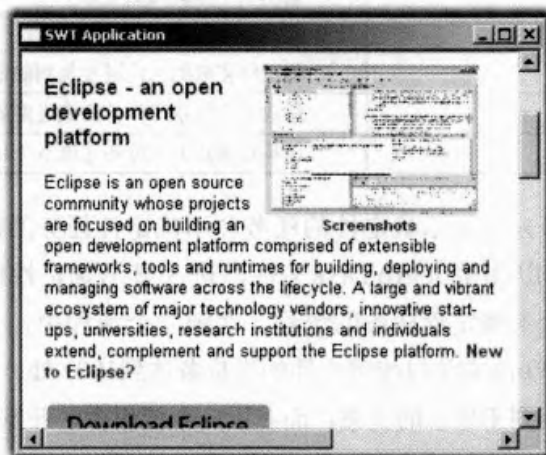


图7-22 使用Browser控件访问网页

SWT的Browser实现使用了OLE技术。OLE (Object Linking & Embedding, 对象链接与嵌入) 是微软制定的一种用于创建复合文档的协议, 任何遵循这个协议的程序所产生的对象都可以被别的对象所包含, 并在需要的时候调用原来的程序激活。如在一篇Word文档中嵌入一个Excel表格, 它在正常情况下显示成一个普通的表格, 但是双击这个表格时, 就可以打开Excel程序来编辑它。使

用OLE技术，程序可以轻松地像搭积木一样将已有的各种功能模块组合在一起构建成功能强大的新程序，由支持OLE技术的程序所创建的文档被称为OLE文档，如Word文档、Excel文档等。

通过SWT提供的OLE相关的类型(org.eclipse.swt.ole.win32.*)，SWT程序可以访问系统中现有的OLE组件并将它们嵌入到自己的界面中。其中比较常用的有以下两个类。

★**OleFrame** —— 所有在SWT界面上使用的OLE部件都必须包含在一个OleFrame中，它负责管理包含在其中的部件尺寸和位置，并在这些部件被激活或不再处于激活状态时更新窗口的菜单栏。可以将它看成SWT_OLE桥一样的角色。

★**OleClientSite** —— 在OleFrame中可以创建OleClientSite，通过OleClientSite可以创建一个新的OLE文档或打开一个现有的OLE文档，它负责将这个OLE文档显示到SWT界面上，并在用户操作文档时与OLE文档对应的处理程序交互。

下面的示例代码演示了使用OLE在一个SWT容器中显示Word界面的情形（代码见光盘：\book.ch7.ole.UsingOLE.java）。

```
public class UsingOLE {
    static OleClientSite clientSite;

    static OleFrame frame;

    public static void main(String[] args) {
        .....

        try {
            frame = new OleFrame(shell, SWT.NONE);
            clientSite = new OleClientSite(frame, SWT.NULL,
                new File("test.doc"));
            createMenus(shell);
            clientSite.doVerb(OLE.OLEIVERB_SHOW);
        } catch (SWTError e) {
            System.out.println("Unable to open activeX control");
            return;
        }
        .....
    }

    protected static void createMenus(Shell shell) {
        Menu menu = new Menu(shell, SWT.BAR);
        shell.setMenuBar(menu);

        MenuItem oleMenuItem = new MenuItem(menu, SWT.CASCADE);
        oleMenuItem.setText("OLE");

        frame.setFileMenus(new MenuItem[] { oleMenuItem });
    }
}
```




```
Menu oleMenu = new Menu(oleMenuItem);
oleMenuItem.setMenu(oleMenu);

MenuItem deactivateItem = new MenuItem(oleMenu, SWT.PUSH);
deactivateItem.setText("Deactivate");
deactivateItem.addSelectionListener(new SelectionAdapter() {

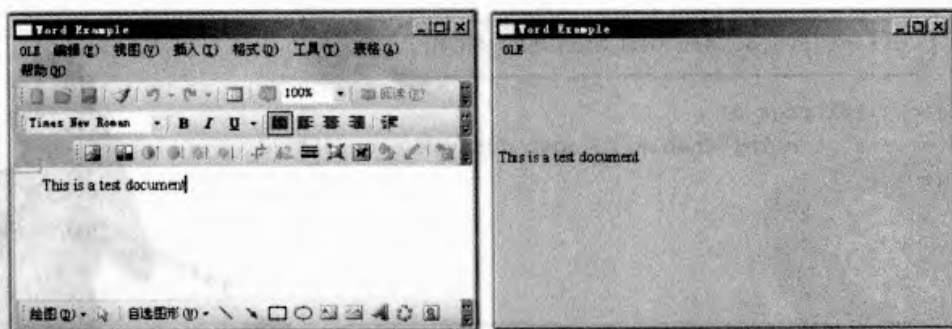
    public void widgetSelected(SelectionEvent e) {
        clientSite.deactivateInPlaceClient();
    }

});
```

代码中标号说明如下：

- ① 在shell中创建一个OleFrame。
- ② 在OleFrame中创建一个OleClientSite，并读入一个Word文档test.doc。OleClientSite会根据文件的扩展名从注册表中查找到对应的处理程序(WinWord)并准备调用它来显示这个文档。
- ③ 为Shell创建一个菜单栏，准备在其中加入一个控制OleClientSite的命令。
- ④ 与通常创建菜单的代码相比，多出了这一条语句。OleFrame会管理它所在的Shell的菜单栏，以便为新创建的OLE文档显示菜单，因此开发者如果想在程序中创建菜单，也需要通知OleFrame才能使菜单显示出来。
- ⑤ 为菜单添加了一个菜单项，单击这个菜单项时，将OleClientSite设为未激活状态。
- ⑥ 调用OleClientSite的方法来激活并显示这个OLE文档(test.doc)。

图7-23是程序的执行效果，程序启动时，会显示Word的编辑界面，用户可以随意修改其中的内容，而当单击OLE菜单中的“Deactivate”将OleClientSite设成未激活状态时，窗口就会只显示一行文字，鼠标单击这行文字会再度回到文字编辑页面。



OLE控件被激活时，显示Word的界面

控件不再处于激活状态，只显示文档的文字内容



图7-23 使用OLE技术在SWT窗口中显示Word文档流程

OLE本身可以提供极为强大的功能，但这项技术毕竟是基于Win32的，使用了OLE技术的程序就失去了跨平台运行的能力。因此在SWT程序中使用OLE组件要三思而后行。

7.6 本章小结

本章介绍了查看器的基本使用方法以及如何使用一些强大的控件，到这一章为止，控件的介绍就告一段落了；下一章将介绍SWT的事件处理机制以及JFace的事件处理框架。

这样的方法真是太简单了，我打赌输给他了！



拉开崭新的学习帷幕

第8章 SWT/JFace的事件处理

控件构成了程序的基础，而事件处理机制则赋予了程序活力。在学习使用各种控件的过程中，读者应该已经对SWT的事件处理机制不再陌生。本章将对这些内容进行总结和深入学习；同时将对JFace所提供的事件处理机制——操作(Action)和贡献(Contribution)加以介绍。通过本章的学习，读者将了解到SWT/JFace的事件处理体系，可以熟练应用各种SWT事件，并能够使用JFace操作与贡献框架管理程序的事件。

本章内容包括：

- ★SWT的事件处理。
- ★常用事件。
- ★JFace事件处理。



进入第08章

8.1 SWT的事件处理

8.1.1 事件处理机制

SWT的事件监听采用了Observer(观察者)的设计模式,在这种模式中,事件的发送者声明了一个监听器接口,对事件感兴趣的各方实现这个接口并将监听器注册到事件的发送者上,在事件发生时,发送者负责向所有注册过的监听器发送事件。

这种模式使得事件的发送方与接受方之间维持了极低的耦合度,因而具有良好的扩展性,也是很多图形库处理事件的模式。图8-1是观察者模式的类图。

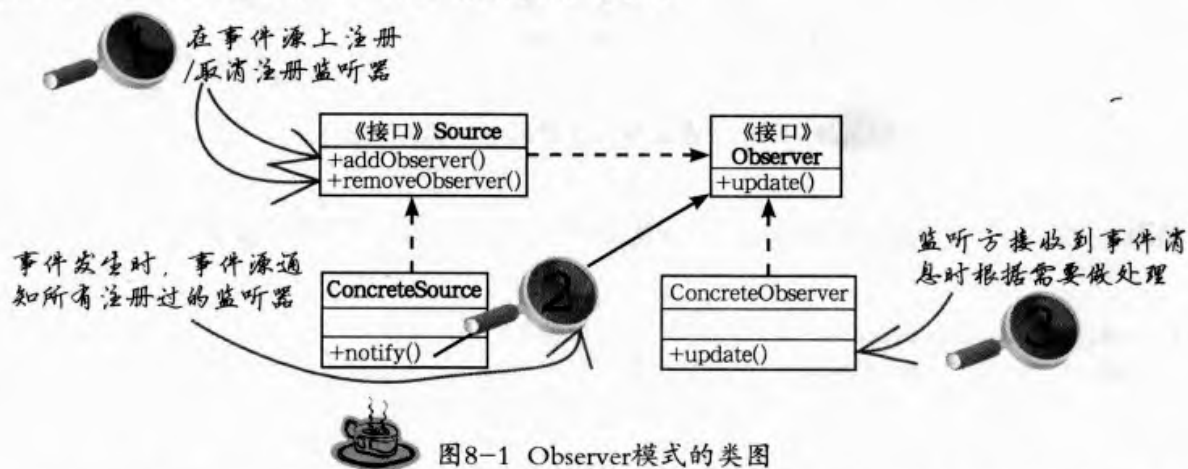


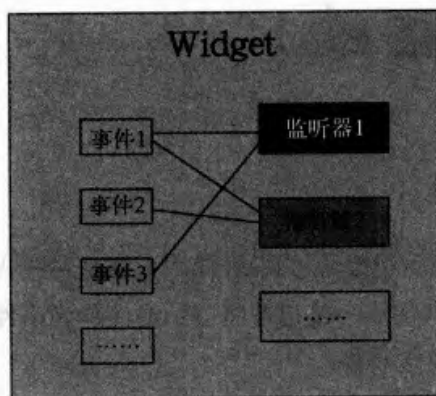
图8-1 Observer模式的类图

在SWT中,Listener接口扮演了观察器接口的角色,而所有窗口组件的父类——Widget类则是事件源。创建了Listener实例后,可以调用Widget.addListener(int eventType, Listener listener)方法将它添加到界面组件上,也可以调用removeListener(int eventType, Listener listener)将它从组件上删除。如向一个按钮添加选择事件监听器可以用下面的代码实现。

```
button.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event event) {
        //处理事件
    }
});
```

上面两个方法中有一个整型类的参数eventType,它代表着Listener所关心的事件类型,如示例中的SWT.Selection就是代表了选择事件的事件类型。Widget上有很多不同的事件可以被触发,出于效率考虑,在某一个事件发生时,Widget并不会通知所有的Listener,而是要求Listener在将自己注册到Widget上面时,必须用eventType来指定它所关心的事件,这样在事件发生时就可以只通知那些关心这个事件的Listener。所有代表事件类型的常量都声明在类SWT中,如SWT.MouseDown代表着鼠标按下,而SWT.KeyDown则代表键盘按键。当某个事件发生时,事件源会调用Listener的handleEvent方法通知监听者。如果某个Listener对多个事件感兴趣,就需要多次调用addListener方法将自己注册到这些事件上,如图8-2所示。

监听器1监听事件1和事件3



监听器2监听事件1和事件2

```
Widget.addListener(事件1, 监听器1);
Widget.addListener(事件3, 监听器1);
Widget.addListener(事件1, 监听器2);
Widget.addListener(事件2, 监听器2);
```

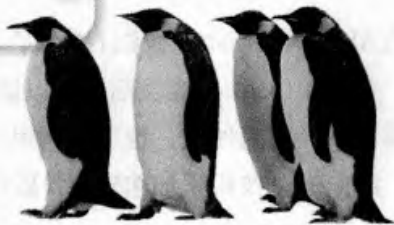


图8-2 将监听器注册到事件源(Widget)



然而，在第4.1节中介绍Button的地方，却是使用如下的代码监听控件的选择事件的。

```
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        //处理事件
    }
})
```



两者的区别在于本节中所介绍的监听器，被称为不分类监听器(Untyped Listener)，它们对所有事件类型均使用同一个方法(handleEvent)监听，而形似SelectionListener或MouseListener等的监听器则被称为分类监听器(Typed Listener)，又称为高级监听器接口。每一个分类监听器各自提供专门监听一种或几种特定事件类型的接口方法，各个窗口组件类对自己能够支持的事件，也会提供支持分类监听器的API，从第3章开始学习控件起，本节之前的示例中一直在使用分类监听器。如Button提供了add/removeSelectionListener方法，而Text则提供了add/removeModifyListener, add/removeVerifyListener等方法。

在SWT内部，所有消息的监听工作都是使用不分类监听器完成的。这种监听器对所有事件都使用统一的数据结构进行维护，因而处理效率比较高，也比较节省内存空间；而分类监听器则是为了方便开发者使用，在不分类监听器的基础上包装过的接口。因为AWT与Swing都使用这种分类的监听器结构，相似的接口结构方便了Swing的程序员转向SWT开发，同时由于分类监听器在名称上就明确地表示了监听器的意图，代码的可读性也比较高。图8-3显示了两种监听器的类型结构。

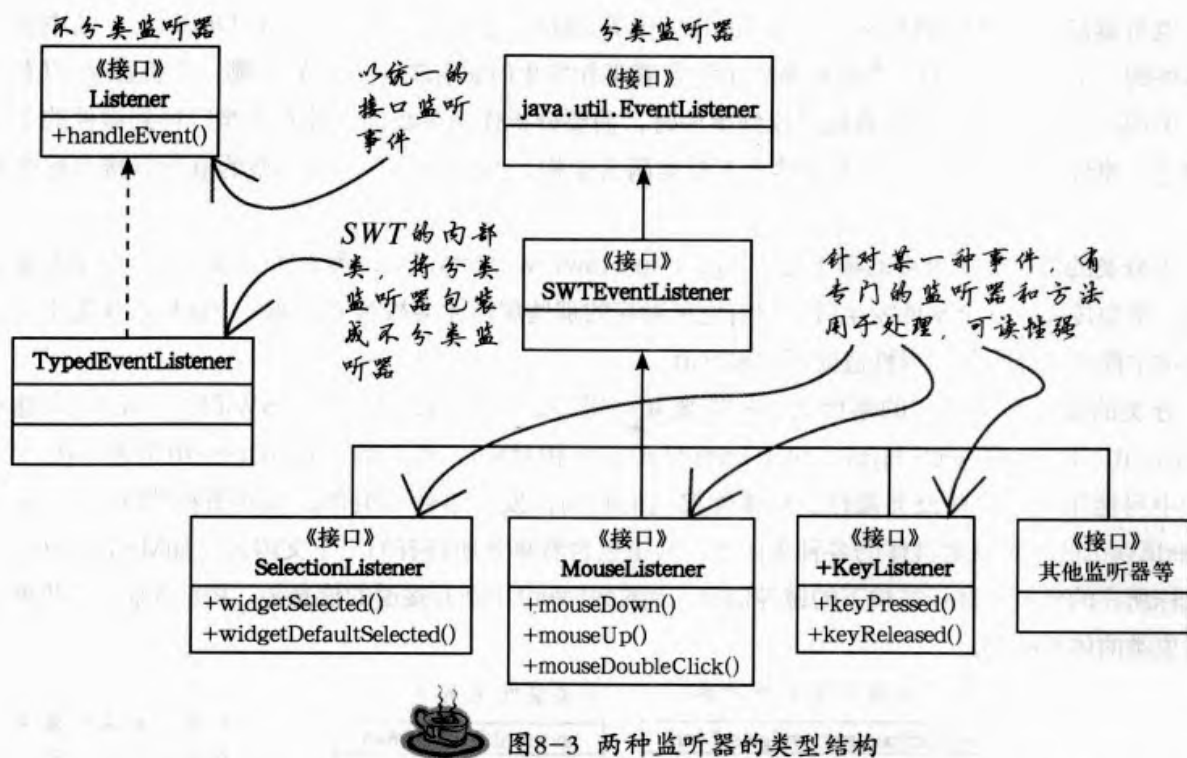


图8-3 两种监听器的类型结构

事实上，在向窗口组件上添加一个分类监听器时，SWT会使用内部类型（所谓内部类型是指这个类仅限于在SWT内部使用，而不应该被应用程序所用）TypedListener将它包装成一个不分类监听器再注册到组件上。当有事件发生时，TypedListener负责将接收到的消息(Event类型)转换成分类监听器的接口所接受的事件类型（如SelectionEvent类型等）并转发给它所包装的分类监听器。图8-4中以Button为例演示了TypedListener是如何在两者之间做转换的。

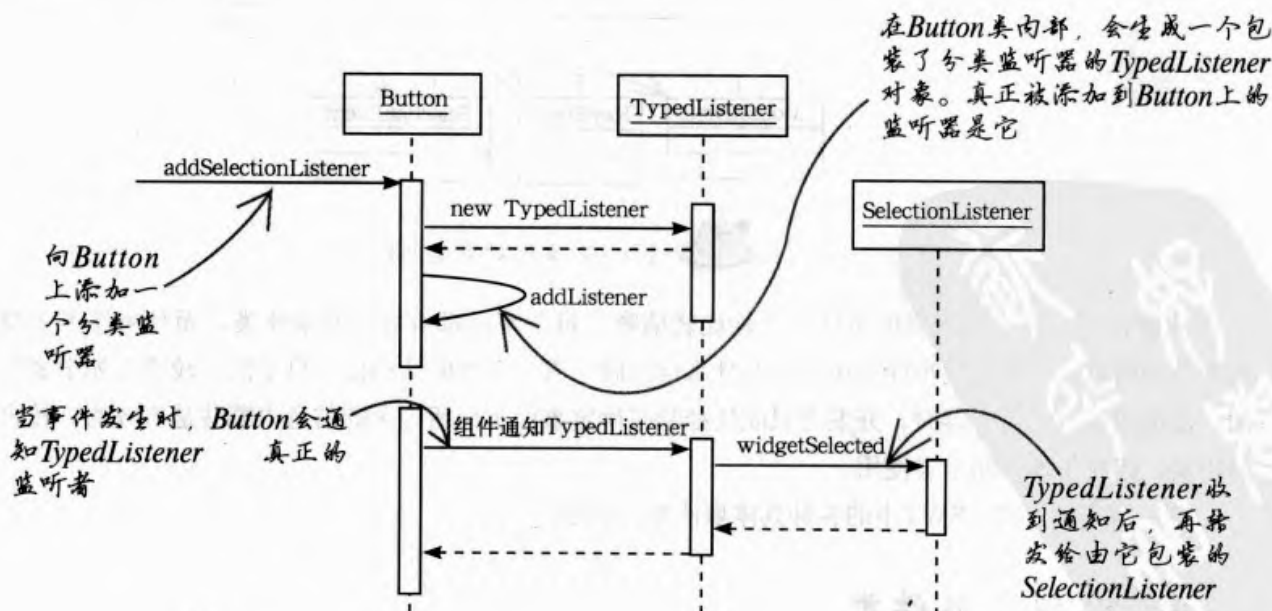


图8-4 TypedListener承担着分类监听器和不分类监听器之间的翻译工作

监听器接口中的参数(Event\SelectionEvent等)被称为事件对象类(Event Object)，其中包含着事件的上下文信息。如一个鼠标事件中包含着事件发生时鼠标的坐标位置、哪一个键被按下(左、右、中)等。监听器通过监听器接口收到事件时，需要分析作为参数的事件对象类以得到事件的上下文信息。事件对象类同样也分为分类与不分类两套架构，它们分别与两种类型的监听器接口配合使用。

不分类的监听器所对应的事件类是org.eclipse.swt.widgets.Event类，所有事件的上下文都要通过这个类型传递。某个事件发生时，只有它所对应的那些属性才会被赋值，如一个鼠标事件发生时，事件类中描述键盘事件的属性值就不会被赋值。

分类的监听器所对应的事件类架构则要复杂得多，它们的公共父类型SWTEventObject继承自java.util.EventObject(与java.util.EventListener相对应)。其子类TypedEvent中包含了在SWT事件中可能用到的一些公共属性，如事件发生的时间，发出事件的组件，事件所在的Display等。TypedEvent的子类就是具体的各种事件类，其中包含着事件所特有的上下文内容。如MouseEvent中有鼠标所在的坐标，鼠标被按下的键等信息；而KeyEvent中则有按键的信息等。图8-5显示了两种事件对象类的体系结构。

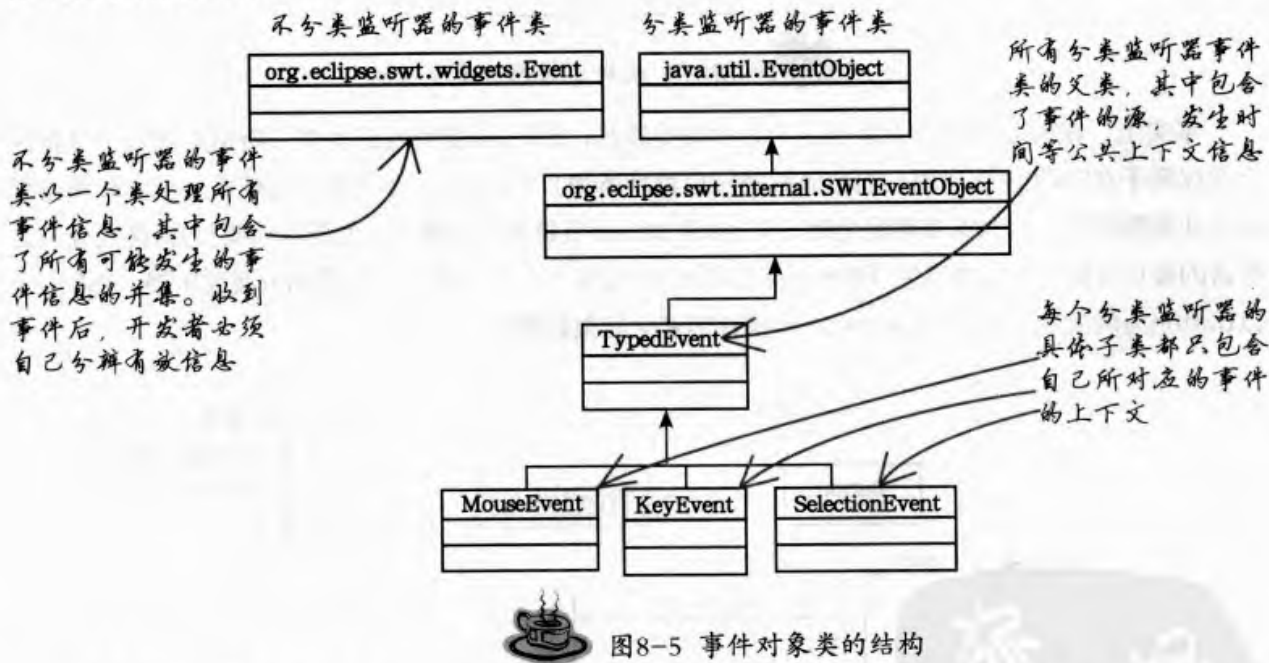


图8-5 事件对象类的结构

两者相比，分类监听器的事件类结构比较清晰，每个事件都有对应的事件类，而且事件类中没有多余的信息，需要知道事件类中包含有什么信息时，看一下类的结构便一目了然，较适合初学者使用；而使用不分类监听器时，开发者只能凭经验来确定事件类中所包含的信息中哪些是有效的，较容易出错，因此不推荐初学者使用。

下一节开始，将对SWT中的各种具体事件加以介绍。

8.1.2 低级事件类

低级事件是指鼠标的进入、单击、拖放或键盘的按键等较为简单的事件，这些事件由操作系统直

接发出。表8-1中列出了SWT中常用的低级事件及相关信息。

表8-1 SWT中的低级事件

事件常量	分类接口名	对应方法	对应的组件	描述
Dispose	DisposeListener	widgetDisposed	Widget	某一个组件的dispose方法被调用
FocusIn	FocusListener	focusGained	Control	焦点落在组件上(鼠标单击或Tab键跳转)
FocusOut		focusLost		焦点离开组件
KeyDown	KeyListener	keyPressed	Control	键盘上某个键被按下
KeyUp		keyReleased		键盘上某个键被放开
MouseDown	MouseListener	mouseDown	Control	鼠标某个键被按下
MouseUp		mouseUp		鼠标某个键被放开
MouseDoubleClick		mouseDoubleClick		鼠标某个键双击
MouseMove	MouseMoveListener	mouseMove	Control	鼠标移动
Paint	PaintListener	paintControl	Control	控件重画(由于内容改变或被其他控件遮挡等原因需要重新绘制控件时)
Move	ControlListener	controlMoved	Control	控件位置(location)改变
Resize		controlResized		控件尺寸(width, height)改变
Iconify	ShellListener	shellIconified	Shell	窗口被最小化
Deiconify		shellDeiconified		窗口被最大化
Close		shellClosed		窗口被关闭
Activate		shellActivated		窗口被激活
Deactivate		shellDeactivated		窗口切换到未激活状态
Show	MenuListener	menuShown	Menu	菜单项被显示
Hide		menuHidden		菜单项被隐藏

低级事件是SWT程序与用户交互的基础，然而在开发过程中却不太常会直接使用它们来控制程序。更多情况下，开发者会直接处理组件上的高级事件。

8.1.3 高级事件类

理论上讲，低级事件涵盖了所有与用户交互的功能，因此使用它们就能够完成对组件的所有操作，但是低级事件功能比较简单，在处理稍微复杂一点的情况时，直接使用低级事件会比较烦琐。因此SWT在低级事件的基础上针对各个组件的具体功能将低级事件进行了包装，这些包装过的事件称为高级事件，每个组件所支持的高级事件各有不同。

如以Text控件为例，开发者可以通过监听键盘事件来得到用户输入的内容。但程序所关注的实际上是控件中文字的改变，因此更好的选择是直接使用Text的Modify事件。这个高级事件将所有导致Text内容改变的因素全都包含了进来，因而使用起来更加方便。

表8-2列出了各个控件中常用的高级事件，其中大部分在介绍对应控件的内容中已经有所提及。

表8-2 SWT中的高级事件

组件	事件	描述	提及章节
Button	Selection	Button被选择(按下)	4.1
Text	DefaultSelection	用户在Text中输入回车键时激发	4.3
	Modify	Text内容已经被改变(原因包括输入字符, 粘贴内容等)	4.3
	Verify	Text内容将要被改变。开发者执行检查后可以决定是否允许这次改变生效	4.3
Combo	DefaultSelection	用户在Combo中输入回车	4.5
	Modify	Combo内容被改变	4.5
	Verify	Combo内容将被改变	4.5
ToolItem	Selection	用户单击工具栏按钮	4.6
MenuItem	Selection	用户单击菜单项	4.7
	Arm	鼠标移动到菜单项上	4.7
	Help	鼠标停在菜单项上时按帮助键	4.7
List	Selection	用户单击List	4.4
	DefaultSelection	用户双击List	4.4
Tree	Selection	用户单击Tree	7.1.5
	DefaultSelection	用户双击Tree	7.1.5
	Expand	树中的某个节点被展开	7.1.5
	Collapse	树中的某个节点被折起来	7.1.5

8.2 常用事件

事件发生时, 组件会通知监听器。但是仅通知监听器“事件发生了”是不够的, 还需要告诉它“什么时候发生”, “在哪里”, 以及其他一些具体内容。这些内容被称为事件的上下文信息。事件的上下文信息是通过事件类传递给监听器接口的。

鼠标和键盘事件是用户与程序交互的基础, 所有事件都是基于它们发生的。本节将对这些常用事件作简单介绍。

8.2.1 鼠标事件

当用户在某个控件的范围内操作鼠标(移动鼠标指针、单击按键等)时, 就会产生鼠标事件, 它的上下文是通过MouseEvent类传递到监听器的。MouseEvent中包含的信息如表8-3所示。

表8-3 MouseEvent中包含的信息

公共信息(每个分类的事件对象类都有这些信息, 定义在TypedEvent中)	
Display display	事件所在的Display对象
Widget widget	发出事件的组件对象
int time	事件的发生时间(以毫秒表示)

续表

鼠标事件的专有信息(与具体事件相关, 定义在各个事件对象类中)	
int x	事件发生时鼠标的横坐标
int y	事件发生时鼠标的纵坐标
int stateMask	事件发生时, 键盘上按下的辅助键(Ctrl, Shift等)
int button	事件发生时鼠标的按键

stateMask(状态掩码)是一个int值, 可以用SWT中代表辅助键的常量与它做按位与操作得到某一个键是否被按下的信息。可用的辅助键值有SWT.CTRL, SWT.ALT和SWT.SHIFT三个, stateMask & SWT.CTRL为0 代表鼠标事件发生时Ctrl键没有按下, 不为0则表示键已按下; stateMask & (SWT.CTRL | SWT.ALT) 不为0则代表Ctrl键和Alt键同时按下。

button值代表鼠标事件发生时被按下的鼠标键。使用三键鼠标时, 代表左键的值为1, 右键为2, 中键为3, 0代表没有键被按下。

8.2.2 键盘事件

当用户按键盘时, 处于活动状态的组件会接收到一个键盘事件。作为上下文的KeyEvent中包含的信息如表8-4中所示。

表8-4 KeyEvent中包含的信息

公共信息	
与MouseEvent中介绍的相同	
专有信息	
char character	键盘事件所包含的按键值, 这个值是将辅助键信息添加到按键信息上得到的
int keyCode	代表了引发本次键盘事件的按键, 键盘上每个键都拥有唯一的keyCode
int stateMask	与KeyEvent中的stateMask意义相同, 代表着这个事件发生时, 键盘上哪些辅助键处于按下状态
boolean doit	标志着这个事件是否有效, 监听器可以将这个值改成false以将事件标识成无效。无效的事件所代表的操作将被取消

键盘事件中有character和keyCode两个值都代表着按键信息。character代表了真正会传递到应用程序的值, 而keyCode则代表着是键盘上哪一个键被按下了。直接按键盘上某一个非控制键(字母、数字等)时, character和keyCode一般是相同的, 但由于辅助键的作用, 同样的character可能会由不同的keyCode所输入。如按住Ctrl键后再按字母键, 就可以输入ASCII码中的控制字符, 如Ctrl+A对应的是0x01的控制符, 而Ctrl+E对应的则是0x05。这样, Ctrl+M的组合键与按Return键的效果是相同的(“\r”控制符), 按Ctrl + M的组合键, 会产生一个键盘事件, 如下所示character = ‘\r’, keyCode = 109(m的ASCII值), stateMask = 262144 (SWT.CTRL)

按回车键时, 产生的键盘事件如下所示。

character = ‘\r’, keyCode = 13(\r的ASCII值)

虽然两个事件输出的字符相同，但不同的keyCode说明一个事件是被Ctrl+M触发的，而另一个则是Return键所触发，如果程序需要区分这两种情况，就可以使用keyCode来判别。

8.2.3 Paint事件

本节将讨论一个平时不太常用到，但是功能十分强大的低级事件——Paint事件。当控件需要被重新绘制时，就会送出这个事件。一般在以下两种情况下，需要重新绘制一个控件。

1. 控件所在的全部或部分屏幕区域曾经被其他窗口遮挡过了，现在需要重新显示它。
2. 程序调用了Control.redraw方法，将控件所在的全部或部分屏幕区域标识为需要重绘。

Paint事件对应的分类监听器PaintListener只有一个方法paintControl，当控件需要被重绘时，会通知所有的监听器。之所以说这个事件强大，是因为从事件类PaintEvent中可以取得用于绘图的图形上下文（Graphical Context, GC）对象，这个对象提供了一系列的用于绘图的方法，使用这些方法就可以在控件的界面上进行任意的图形绘制工作。

下面的示例代码为一个Shell添加了PaintListener，在paintControl方法中，在Shell里面画了一根由左上到右下的斜线（代码见光盘：\book.ch8.event.UsingPaintEvent.java）。

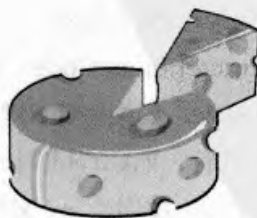
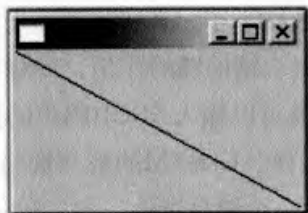
```
shell.addPaintListener(new PaintListener() {
    public void paintControl(PaintEvent e) {
        Shell se = (Shell) e.widget;

        Rectangle bounds = se.getClientArea();

        GC gc = e.gc;
        gc.drawLine(bounds.x, bounds.y, bounds.x + bounds.
            width,
            bounds.y + bounds.height);
    }
});
```

调用Shell的
getClientArea方法，得到了Shell
中除去标题栏部
分，真正可以用于
绘图的区域

使用GC的drawLine方法，
在Shell中画了一根线



代码中使用GC.drawLine方法绘制了一根实线。GC提供了丰富的方法用于图形绘制，表8-5列出了部分常用的方法。

表8-5 GC中常用的绘图方法

方法名	说明
drawPoint	在指定的位置画一个点
drawLine	从指定的起点到终点画一条线
drawRectangle/fillRectangle	根据指定的范围画/填充一个矩形
drawPolygon/fillPolygon	连接所给出的各个顶点画/填充一个多边形
drawOval/fillOval	根据参数画/填充一个椭圆区域
drawText	在给定的位置显示一行文字

默认情况下, GC在绘图时会使用黑色、宽度为1的线条, 填充区域时也会使用系统默认的背景色(如XP系统在经典风格下, 默认背景色为灰色)。如果希望使用其他的线条颜色或填充颜色, 需要在绘图操作之前设置画笔的信息。表8-6列出了设置这些信息的常用方法。

表8-6 设置画笔信息的方法

方法名	说明
setBackground	设置背景色。背景色将用于填充操作及绘制文字时填充文字背景
setForeground	设置前景色。绘制线条时将使用前景色, 绘制文字时, 文字也将被绘成前景色。
setLineWidth	设置线条的宽度, 以像素为单位。默认为1像素
setLineStyle	设置线条类型, 常用类型有SWT.LINE_SOLID: 实线; SWT.LINE_DASH: 虚线; SWT.DOT: 由间隔的点构成的线等

灵活应用这个事件, 可以设计出一些具有特殊显示效果的控件, 比如随着进度的增加, 颜色逐渐变化的进度条等等。

8.2.4 应用举例

恰当使用各种事件, 可以创建出许多有趣的SWT程序。下面的例子演示了如何监听鼠标事件, 从而实现在Shell中绘制线条(代码见光盘: \book.ch8.event.DrawLineInShell.java), 程序效果如图8-6所示。

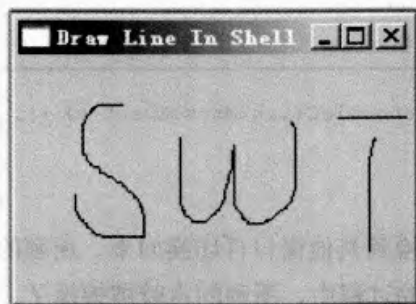
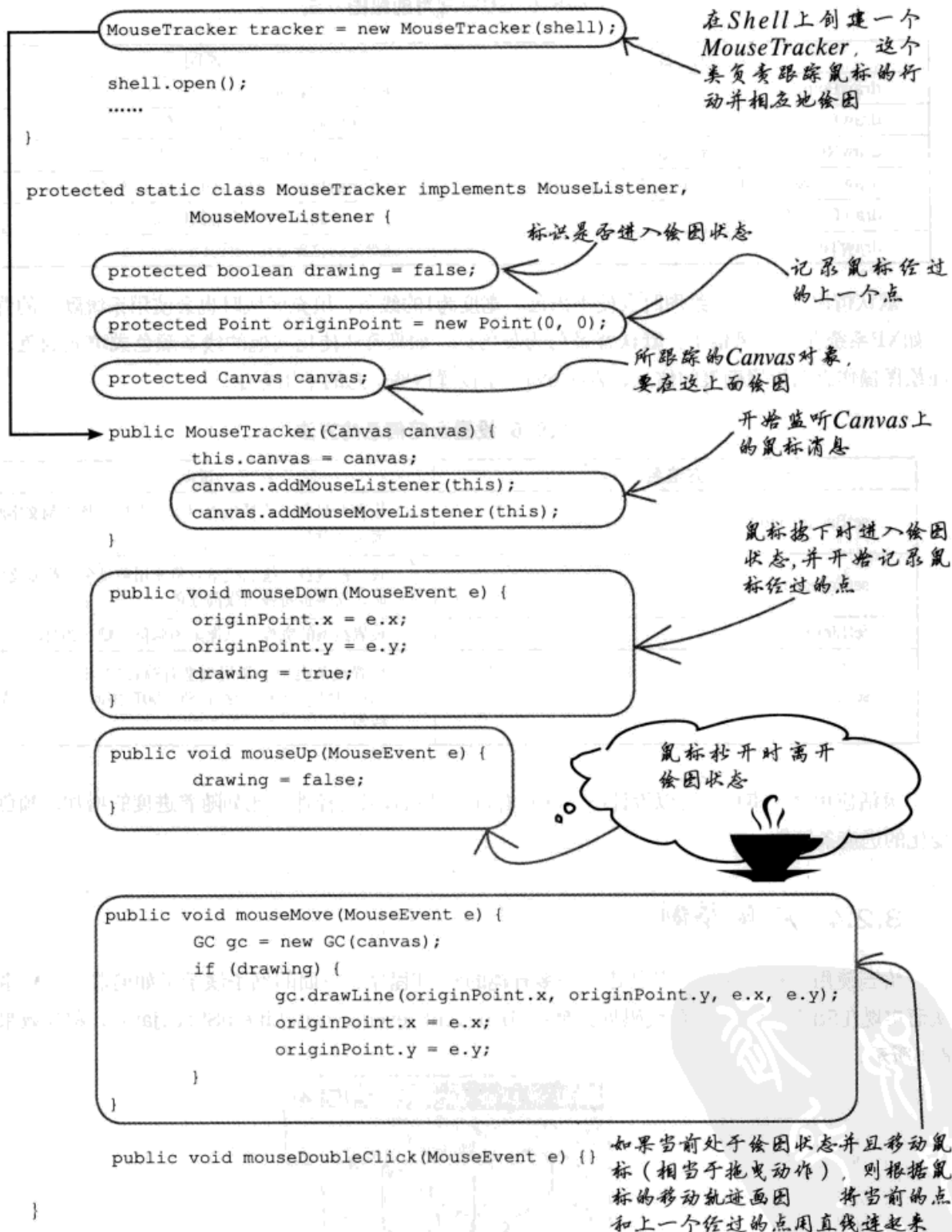


图8-6 在Shell中画线的示例运行效果





在窗口中画了线条后, 如果切换到其他窗口再切换回来, 所画的线条会消失。这是因为每次窗口重新显示时都会重绘自己, 在重绘的过程中, 所画的点就被擦掉了。为了使得这些画痕不被擦掉, 当用户使用鼠标画图时, 需要将所画过的点保存下来, 然后监听窗口的Paint事件 (当窗口重绘时, 会发送这个事件), 并在处理这个事件时, 将点重新画到屏幕上。这一步作为练习请读者自己尝试完成。

8.2.5 使用SWT模拟键盘/鼠标事件

在SWT中，可以使用代码模拟最终用户的鼠标、键盘操作，开发者首先创建一个描述这些操作的事件(org.eclipse.swt.widgets.Event)实例，并使用Display.post()将它加到操作系统的事件队列中，程序就会把这个事件等同于真正来自操作系统的事件加以处理，这个模拟功能在开发图形界面程序的自动化测试时有很大的用处。

下面的代码演示了如何利用该功能模拟键盘事件，窗口打开后，程序会将焦点自动定位到窗口中的文本框并输入一行文字(代码见光盘：\book.ch8.event.SimulateEvent.java)。

```
.....
final Text text = new Text(shell, SWT.BORDER);
text.setBounds(10, 10, 100, 20);
```

```
new Thread() {
```

```
    public void run() {
```

```
        try {
```

```
            Thread.sleep(2000);
```

```
        } catch (InterruptedException e1) {
```

```
            e1.printStackTrace();
```

```
        }
```

```
        display.syncExec(new Runnable() {
```

```
            public void run() {
```

```
                text.setFocus();
```

```
            }
```

```
        });
```

```
String string = "Love the method.";
```

```
for (int i = 0; i < string.length(); i++) {
```

```
    char ch = string.charAt(i);
```

```
    boolean shift = Character.isUpperCase(ch);
```

```
    ch = Character.toLowerCase(ch);
```

```
    if (shift) {
```

```
        Event event = new Event();
```

```
        event.type = SWT.KeyDown;
```

```
        event.keyCode = SWT.SHIFT;
```

```
        display.post(event);
```

```
    }
```

```
    Event event = new Event();
```

```
    event.type = SWT.KeyDown;
```

```
    event.character = ch;
```

```
    display.post(event);
```

创建一个新线程，负责发送模拟事件的工作

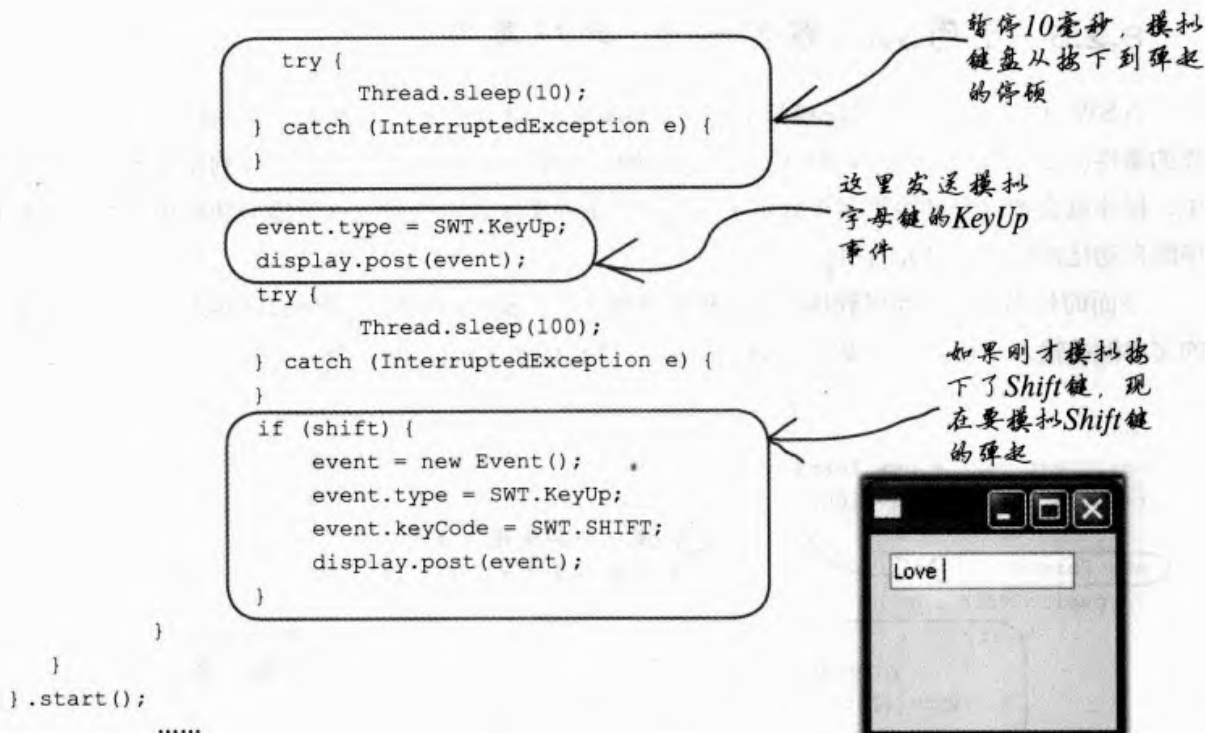
暂停2秒钟后开始模拟操作

首先将焦点定位到Text控件。这样后续的KeyEvent才会发送到Text中

如果当前要发送的字母是大写字母，首先发送一个Shift键被按下的事件

发送代表字母的按键事件





在模拟键盘操作时，主程序另外生成了一个线程负责生成KeyEvent，并通过Display的asyncExec功能与UI线程同步。

模拟一次按键操作由两个KeyEvent构成——一个KeyPressed事件和紧接着的一个KeyReleased事件。如果需要输入大写字母，那就要先发送一个Shift键的KeyPressed事件以模拟按下Shift键，然后发送按字母键的事件，最后再发送Shift键的KeyReleased事件完成整个过程，模拟其他功能键——Ctrl键或Alt键的过程也与之类似。

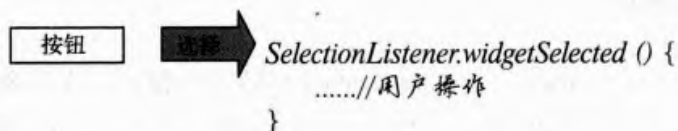
在这个实例中可以看到，模拟鼠标、键盘事件的功能固然强大，但操作比较烦琐，而且要求开发者对KeyEvent, MouseEvent等的结构十分熟悉，在日常的开发工作中，通常是不需要用到这个高级功能的。

8.3 JFace事件处理

8.3.1 操作 (Action) 与贡献 (Contribution)

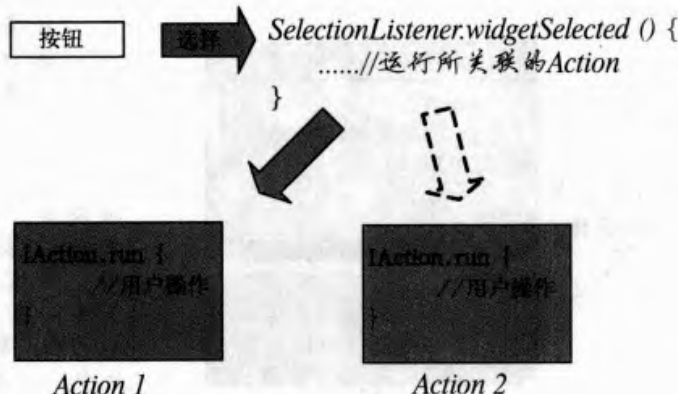
操作是JFace在SWT的事件监听框架基础上，对UI操作方法做出的更高一层的封装。使用操作，可以将一系列用户自定义的命令封装成一个对象，开发者可以将操作关联到界面中的一个或多个组件上，当用户操作其中某个组件时，操作中所包含的命令就会被执行。这种关联是动态完成的，可以在运行时改变，这就将用户操作和具体的界面控件分离开来。使程序具有更大的灵活性，图8-7显示了两者的区别。

传统的SWT事件模型



在运行时很难动态改变用户操作的内容

使用操作/贡献框架



可以在运行时改变所关联的Action,也就随之改变了所要执行的用户操作



图8-7 直接监听SWT事件与使用操作的对比

操作中除封装了用户操作外,还包含了它本身应该如何显示在界面上的信息,如图像、文字、工具提示(Tooltips)等信息,这可以保证同一个操作在不同的组件上有一致的外表,而贡献则负责将操作和具体的SWT组件(ToolItem,MenuItem等)关联起来。

贡献分为两部分,贡献项目(ContributionItem)和贡献管理器(ContributionManager)。贡献项目关联着操作,而贡献管理器则包装了工具栏、菜单等可以放置操作的控件。当开发者将一个贡献项目添加到贡献管理器上时,贡献管理器会从贡献项目中取出对应的操作的显示信息,并生成一个新的组件(工具栏按钮、菜单项等)显示在界面上。图8-8演示了贡献管理器、贡献项目和操作三者之间是如何互动的。

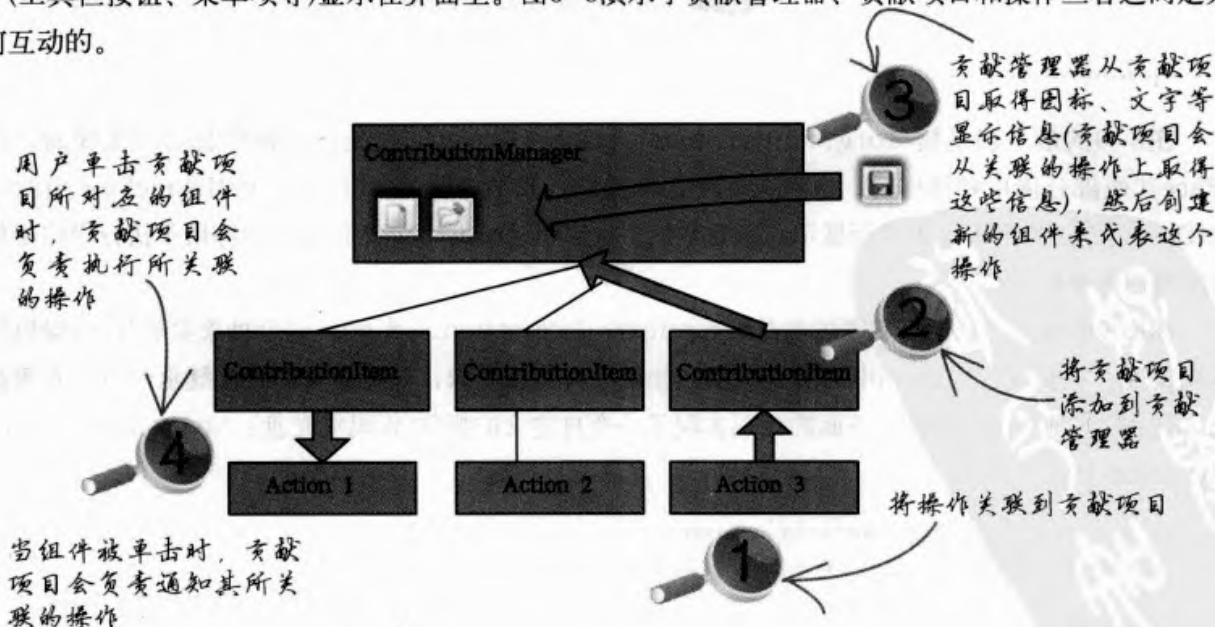


图8-8 贡献项目和贡献管理器

8.3.2 创建操作

所有的JFace操作都要实现org.eclipse.jface.action.IAction接口，这个接口规定了操作需要实现的一般方法，如运行用户操作，提供图片、文字信息等，图8-9显示了JFace为操作提供的继承结构。

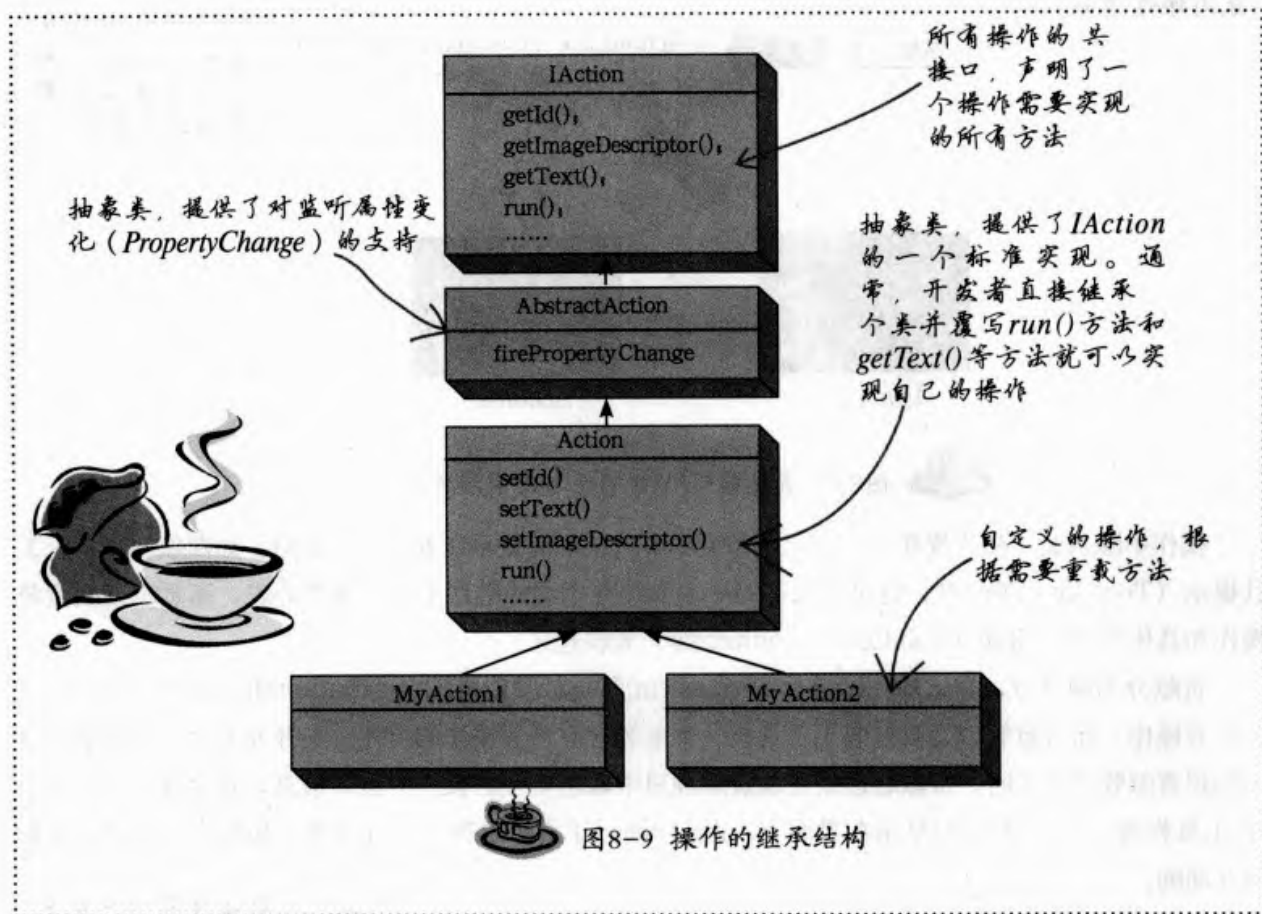


图8-9 操作的继承结构

图8-9的最上层是接口org.eclipse.jface.action.IAction，这是所有操作都必须实现的。在JFace其他部分使用到操作时，也只会使用这个接口。其中声明了getText，getImageDescriptor，getDescription等用于提供界面显示内容的方法，也声明了run，runWithEvent等用于执行操作中包含的用户命令的方法。

图8-9中另外一个比较重要的类是org.eclipse.jface.action.Action，这个抽象类是JFace提供的标准实现，它实现了IAction中的所有方法。如果没有特殊需求，开发者可以直接继承Action并重载run方法来实现自己的操作。下面的代码实现了一个自定义的操作(代码见光盘：\book.ch8.action.MyAction.java)。

```
public class MyAction extends Action {
    public static String ID = "MyAction";

    public MyAction() {
        super();
        setId(ID);
    }
}
```

```

    setText("My Action");
    setToolTipText("My Action Tooltip");
    setImageDescriptor(ImageDescriptor.createFromFile(this.
getClass(),
"toolbar1.gif"));
}

public void run() {
    System.out.println("My Action Run");
}

```

在构造函数中设置各种用于显示的内容

重载run方法以实现自定义的用户操作

在下面一节的示例中将多次使用到这个操作对象，它的图标是写字板的图片，运行时会在控制台中打出“My Action Run”的字样。

8.3.3 使用贡献

本节将概述如何使用贡献框架完成一些常见的任务，如生成工具栏按钮、生成菜单项等。

贡献框架将绝大部分实现内容都包在了内部，因此使用它们进行开发就变成了一件极为简单惬意的事情。下面三行代码基于一个已有的工具栏创建了一个ToolBarManager，并向上面添加了一个操作(代码见光盘：\book.ch8.action.UsingToolBarManager.java)，生成的工具栏如图8-10所示。

```
ToolBarManager toolBarMgr = new ToolBarManager(toolBar);
```

```

toolBarMgr.add(new MyAction());
toolBarMgr.update(true);

```



图8-10 使用ToolBarManager控制工具栏

代码中标号说明如下：

这段程序向ToolBarManager中添加了一个操作后，又调用了它的update方法，这是因为ToolBarManager所对应的工具栏已经被创建出来，出于效率考虑，ToolBarManager不会立刻为每一个新添加的操作生成对应的工具栏按钮，那么为了让刚添加的操作显示出来，就要调用ToolBarManager的update方法强迫立刻刷新整个工具栏。

下面将尝试使用MenuManager管理窗口菜单。由于菜单结构比工具栏来得复杂，使用MenuManager管理菜单比ToolBarManager也略微麻烦一些。另外，与ToolBarManager不同，MenuManager不能基于已有的Menu控件创建，而必须遵循“先生成管理器，再从管理器中生成菜单”的顺序。下面的代码创建了一个管理着窗口菜单栏的MenuManager，并向其中添加了一个操作和一个子菜单（代码见光盘：\book.ch8.action.UsingMenuManager.java），程序运行效果如图8-11所示。

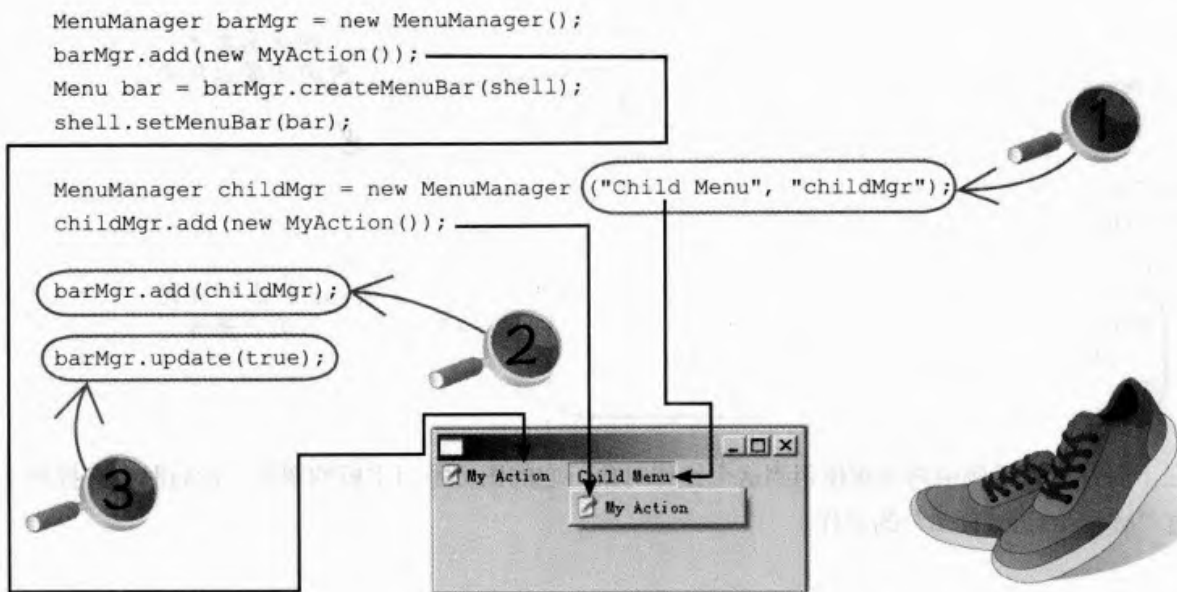


图8-11 使用MenuManager控制菜单

代码中标号说明如下：

- ❶ 关于创建MenuManager时提供的两个String参数，第一个决定了该MenuManager作为子菜单时，对应的MenuItem的显示文字；第二个则是该MenuManager唯一的ID，在父MenuManager中，可以通过find(String id)这个方法找出指定ID的子MenuManager。
- ❷ MenuManager实现了IContributionItem接口，因此可以作为一个贡献项目添加到父MenuManager中。
- ❸ 和前面处理ToolBarManager时的情况类似，由于向父MenuManager中添加这个子MenuManager时，菜单栏已经生成了，因此需要调用父MenuManager的update方法刷新它的内容才能使新添加的菜单显示出来。

8.4 本章小结

本章系统地介绍了SWT中的事件处理框架结构。还介绍了JFace对事件处理进行的包装——操作和贡献。在Eclipse插件开发中，普遍采用了操作与贡献的思想。

到本章为止，读者应当已经掌握了较为全面的SWT基础知识。从下一章开始，将结束对SWT的学习，正式进入Eclipse插件编程的学习阶段。



第9章

Eclipse 插件体系结构

拉开崭新的学习帷幕

本书的第1章简要介绍了Eclipse的插件技术，从本章起将开始讨论同插件开发相关的内容。本章将讨论Eclipse的插件体系结构，从整体上概览整个Eclipse，为后面的深入介绍奠定基础。通过本章的学习，读者将熟悉Eclipse的平台架构、工作模式，插件之间的相互组织方式等。

本章内容包括：

- ★ Eclipse平台体系结构。
- ★ 插件的加载过程。
- ★ 插件的扩展模式。

进入第09章



9.1 Eclipse体系结构

Eclipse不是封闭的程序，可以说是众多“可供插入的地方”（在Eclipse中称为扩展点）和“可以插入的东西”（在Eclipse中称为扩展）的结合。这种架构使Eclipse非常易于扩展。现实生活中有很多这样的例子，图9-1显示了电脑中连接外设的多种插口，这里的插口可以看做是计算机的“扩展点”。计算机通过这些扩展点连接各种各样的外部设备，以满足不同应用的需要。尽管这些外设有不同的外形，不同的用途，但它们必须能够插入计算机的“扩展点”中，也就是必须满足计算机所规定的接口。



图9-1 计算机连接外设的多种插口

9.1.1 Eclipse平台架构

如图9-2所示为Eclipse的体系结构图，其中间部分为Eclipse的平台部分。Eclipse是围绕插件概念构建的，Eclipse平台部分的所有子系统除了很小的核心之外，都由插件构成。

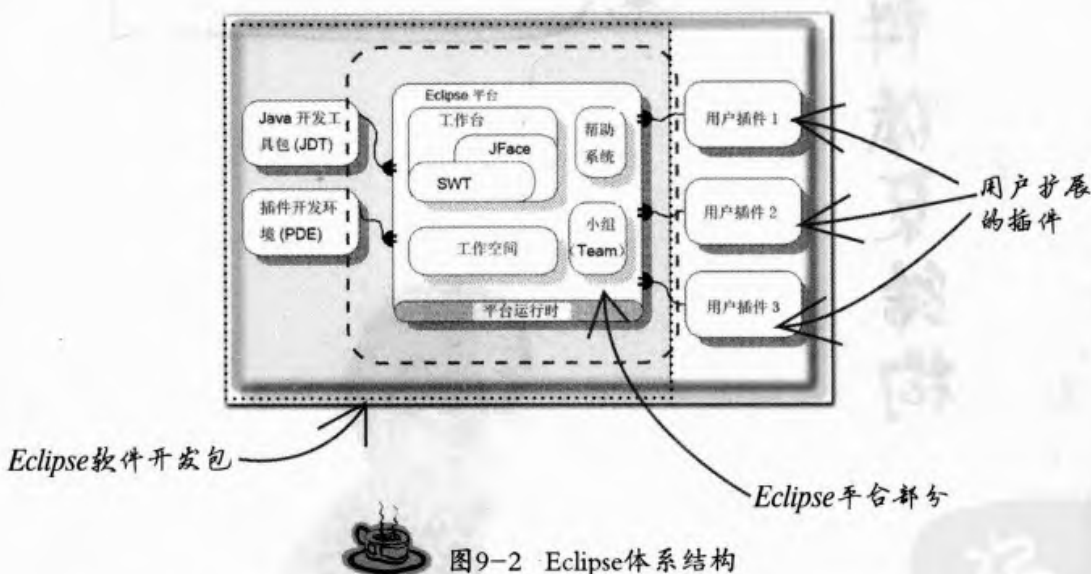


图9-2 Eclipse体系结构

Eclipse平台中的每个子系统本身是由实现某些关键功能的一组插件构建的。这些插件包括内容如下所示。

- ★运行时(Runtime)——运行时定义了插件的结构，负责在Eclipse运行时发现和管理插件。
- ★工作空间(Workspace)——工作空间定义了Eclipse的工作区，用来分配和跟踪资源。
- ★标准窗口小部件工具箱(SWT)——一个UI窗口小部件通用库，它能生成本地化风格的用户交互界面组件，但是拥有独立于操作系统的API。
- ★Face——建立在SWT之上，提供对常用UI任务的支持。
- ★工作台(Workbench)——工作台定义了Eclipse的UI聚合体，提供包括编辑器、视图、透视图等界面组成部分。

★小组 (Team) —— “小组” 插件为Eclipse中集成各种类型的源代码控制管理系统 (如CVS,Subversion等) 提供服务。

★帮助 (Help) —— “帮助” 插件为开发者编写帮助文档提供支持。

运行时和工作空间组成了Eclipse的核心部分。通常所说的Eclipse插件便是向基于Eclipse核心的系统提供服务的组件。在Eclipse SDK中,除了Eclipse平台以外,还包括了Java 开发工具 (JDT) 和插件开发环境 (PDE) 两个组件来支持插件开发。它们各自的作用如下所示。

★JDT: JDT提供对Java程序的编辑、编译、调试、测试功能,实现功能完整的Java开发环境。

★PDE: PDE提供自动创建、处理、调试和部署插件的工具。

Eclipse平台和JDT、PDE构成了Eclipse的三层结构,如图9-3所示。

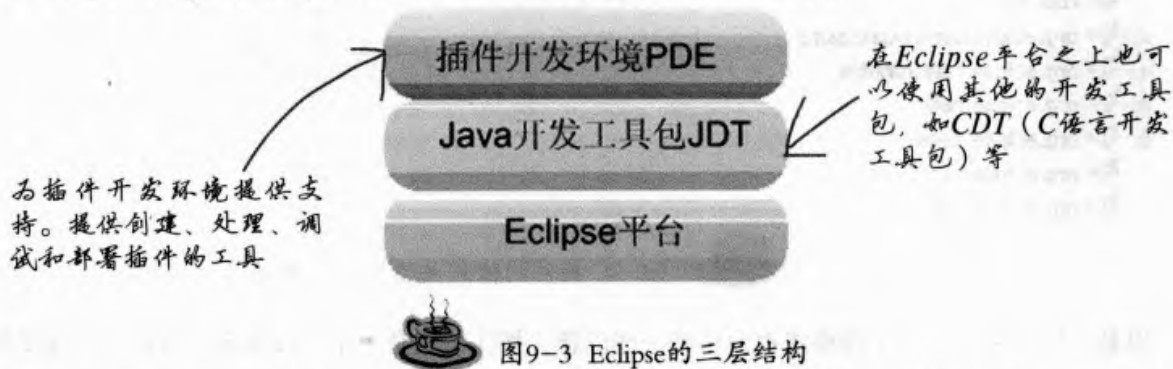


图9-3 Eclipse的三层结构

在三层结构之上,用户可以遵循Eclipse平台定义的插件结构自行编写插件来扩展Eclipse平台的功能。维护Eclipse的组织还为开发者分离出了Eclipse核心,开发者也可以基于Eclipse核心开发自己的应用程序。

9.1.2 插件工作模式

Eclipse将各个模块分成了离散的功能部件,那么,各个插件是以什么样的模式工作的呢?

Eclipse通过插件的依赖关系将不同的插件联系在一起。每个插件可以扮演双重角色,即其他插件服务的使用者和其他插件所需服务的提供者。以Eclipse UI插件的依赖关系为例 (箭头指向的是被依赖的插件),如图9-4所示,图中Help插件依赖于Runtime插件提供的服务,又提供了Workbench插件需要的服务。Eclipse的这种工作模式为程序员提供了一个开放的工作环境,有助于快速开发出功能强大的应用程序。



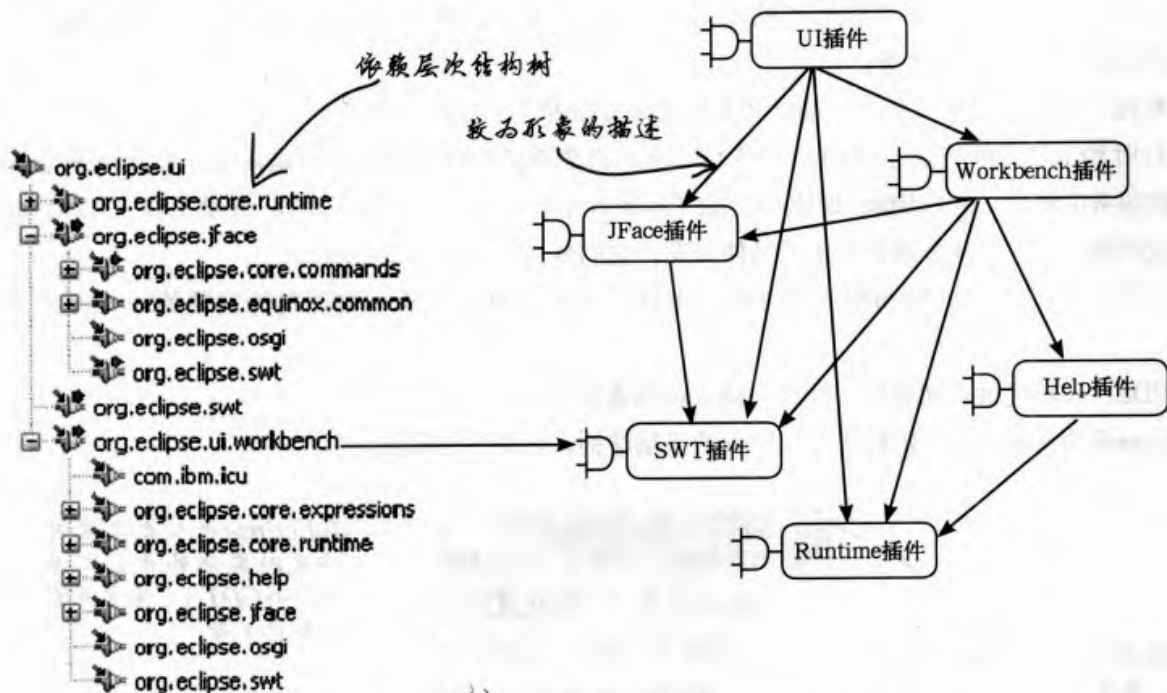


图9-4 插件的依赖关系的一个例子

但是，Eclipse的这种工作模式也会带来一些问题。插件模式是一个扩展体系，随着插件规模的越来越多，Eclipse运行时所消耗的资源会迅速增长，直到耗费系统中的所有资源。为了避免这一情况发生，Eclipse使用了懒加载的工作方式，运行时的总体目标是最终用户不应该为已安装但未使用的插件消耗内存或性能。平台扩展模型允许运行时引擎确定哪些扩展点和扩展已经由插件提供了但是目前为止还没有运行过。这样，可以安装许多插件，但是将不会在启动的时候就激活它们，直到根据用户的活动请求了插件所提供的功能为止。

※ 注意 : ※

Eclipse的懒加载模式延迟了资源过度耗费现象的发生，但并没有避免。因为在3.2及以前的版本中，Eclipse虽然提供了插件的动态卸载功能，但并不总是有效。随着用户请求更多的功能，内存被使用的资源还是会增加。



9.1.3 工作台层次结构

在构建插件之前，应该首先熟悉一下Eclipse工作台的层次结构。Eclipse工作台是不可见的，它包括一个或多个Eclipse工作台窗口（IWorkbenchWindow）。运行Eclipse之后，出现如图9-5所示的界面，其中顶层窗口就是Eclipse的工作台窗口，每个工作台窗口又包括菜单栏、工具栏和多个工作页面（IWorkbenchPage）。主要的工作台部件——视图（View）和编辑器（Editor）就包含在工作台页面中。

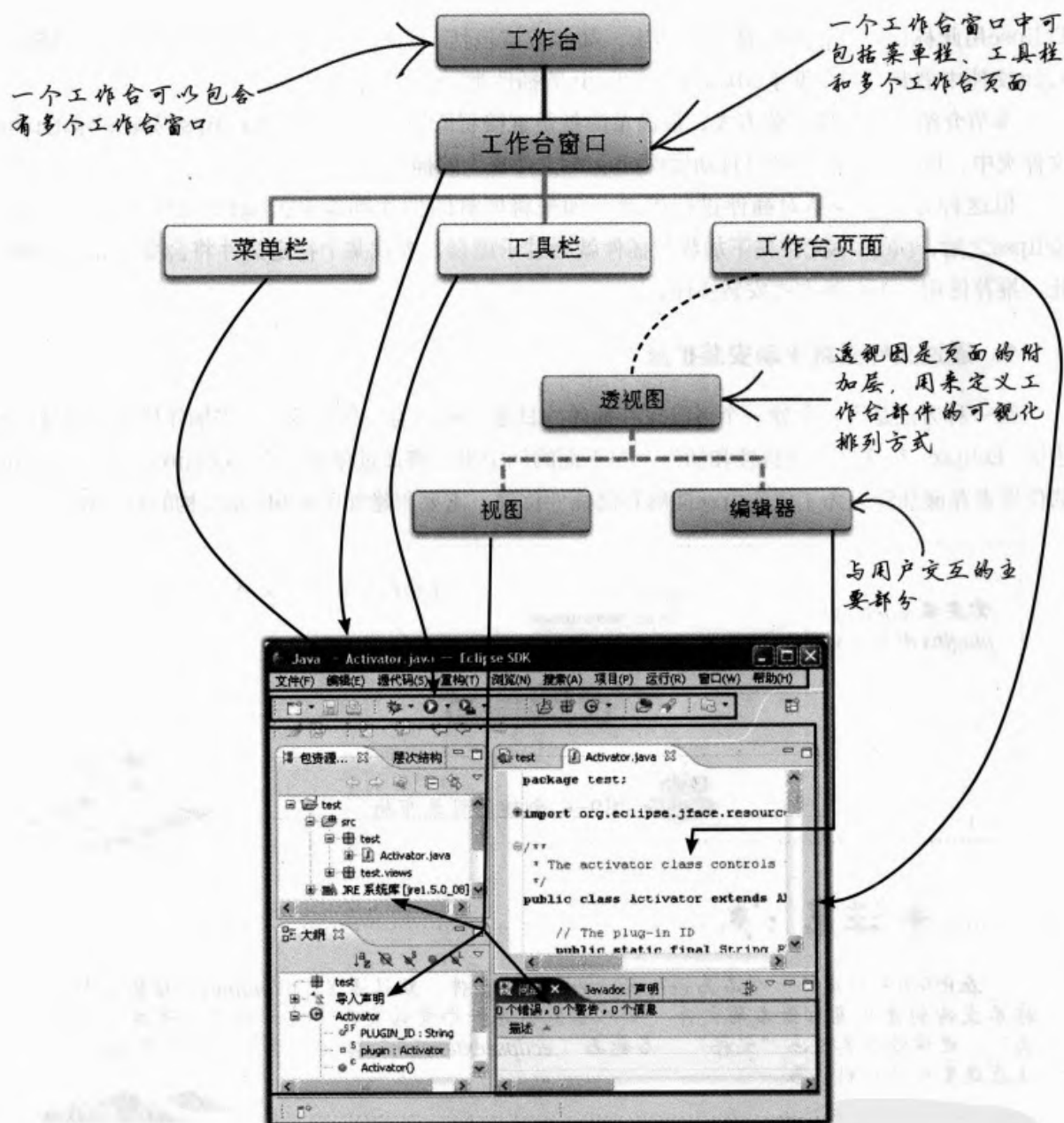


图9-5 Eclipse工作台层次结构及对应界面元素

9.2 插件的加载过程

要使Eclipse加载某个插件需要经过三个过程，插件的安装、插件的查找和插件的启动。

9.2.1 插件的安装

Eclipse的插件必须满足几个条件才能被Eclipse安装。首先，插件必须有一个唯一的标识符，

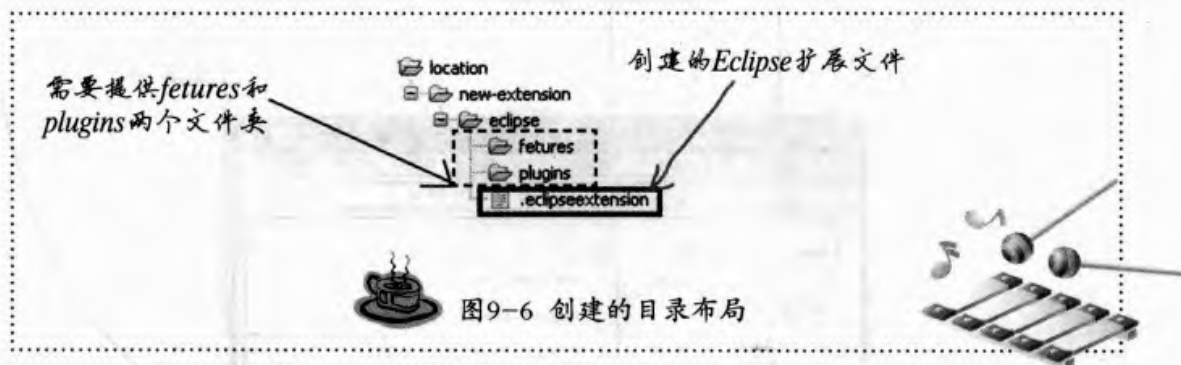
Eclipse用此标识符来识别此插件；其次，插件应该包括 *.jar文件、清单文件以及所需的资源，在 *.jar文件中需要有一个继承AbstractUIPlugin的插件类。

本节介绍三种插件安装方式，最简单的就是直接将Eclipse的插件放入Eclipse安装目录的plugins文件夹中，Eclipse会在启动时自动发现plugins文件夹中的插件。

但这种方式不容易对插件进行管理。如果将所有的插件都置于plugins文件夹下，使用一段Eclipse之后，plugins文件夹下加载的插件就会过于庞杂，查找某个特定插件将会变得非常困难。因此，推荐使用以下两种方式安装插件。

1. 通过文件系统手动安装扩展

第一种方法是手动创建一个可以保存插件的目录（称之为产品扩展），将插件移到该目录，然后告诉 Eclipse 在这里寻找特性和插件。在下面的例子里，将通过创建一个/location/new-extension的位置来存储插件。为了使Eclipse能够存储插件位置，需要创建如图9-6所示结构的目录和文件。



※ 注意 : ※

在图9-6中创建了一个名为.eclipseextension的文件，默认情况下Windows平台的文本文档不支持创建只有后缀名的文件，可以首先创建一个空的文本文档，打开后选择“另存为”，选择保存类型为“文档”，名称为“.eclipseextension”保存。这个文件让 Eclipse 知道在这里可以找到扩展。

在.eclipseextension文件中输入如下所示的内容。

.eclipseextension 文件中的 version 属性应该设置为使用这个产品扩展的 Eclipse 的版本。本例中使用的是Eclipse 3.2.1。

```
id=org.eclipse.platform name=Eclipse Platform
version=3.2.1
```

下一件事是将这个插件位置告诉 Eclipse，以便它以后在这里查找插件。在Eclipse中选择“帮助 (Help)” → “软件更新 (Software Update)” → “管理配置 (Manage Configuration)” 得到“产品配置 (Product Configuration)”，选中Eclipse SDK，右键选择“添加” → “扩展位置”，如图9-7所示。

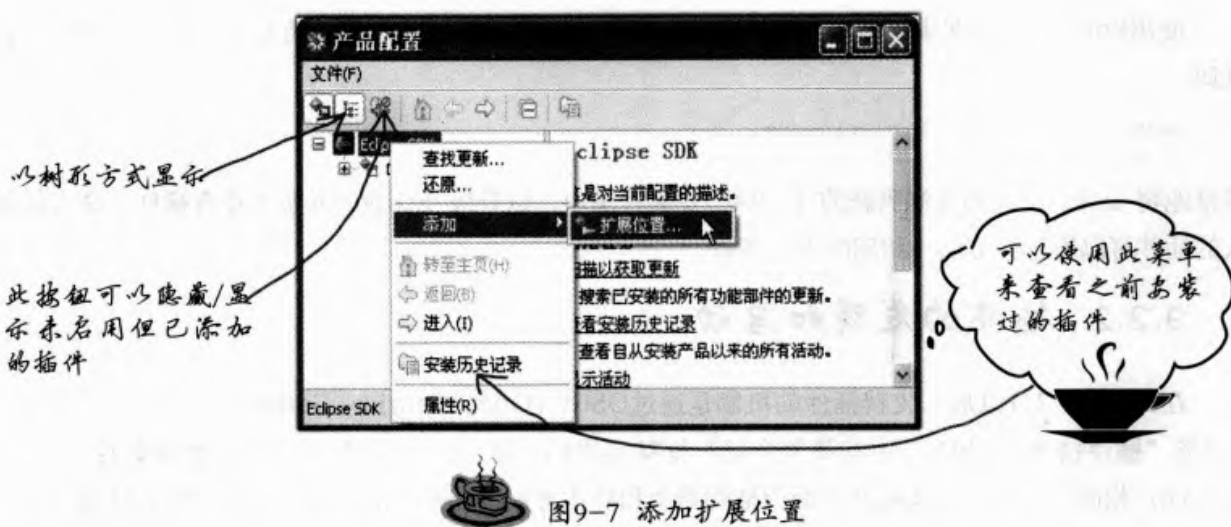


图9-7 添加扩展位置

新的扩展位置将会出现在产品配置里。如果在新扩展位置中的plugins文件夹中已经放入插件，这时只要重启Eclipse，新的插件就会生效。这种方法非常好的一点是，启用/禁用插件非常方便，如图9-8所示。

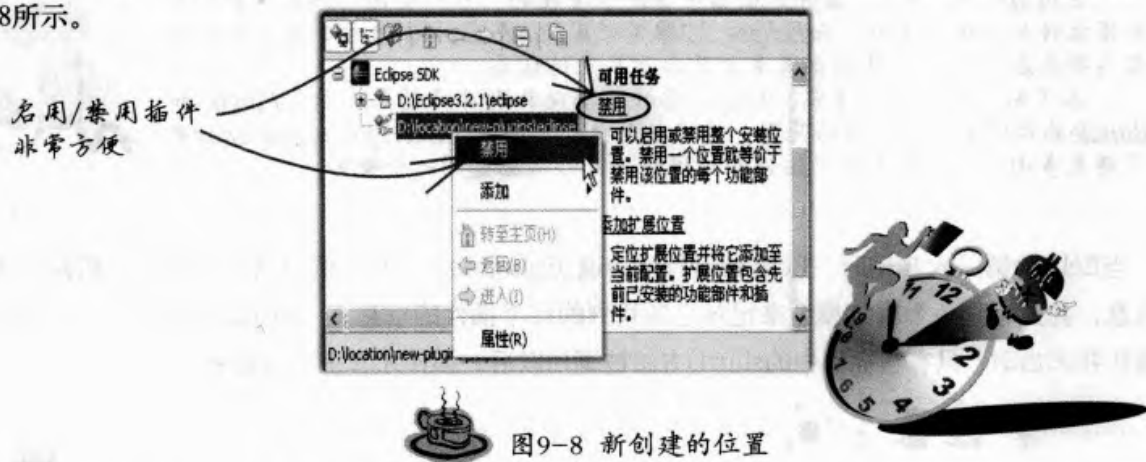


图9-8 新建的位置

2. 采用链接的形式安装插件

如果在文件系统中已经有了产品扩展，例如已经使用上述方法创建了扩展，就可以在 Eclipse 程序目录中创建一些简单的文件，告知 Eclipse 需要检查这些目录以寻找相应的插件。

在 Eclipse 安装文件夹中创建一个名为 links 的目录。在这个目录中，创建 *.link 文件（例如 myPlugins.link）。每一个链接文件指向一个产品扩展位置。Eclipse 会在启动时扫描这个 links 文件夹，并在每个链接文件所指向的产品扩展中寻找插件。图9-9显示了一个使用 link文件夹的 Eclipse 安装布局。

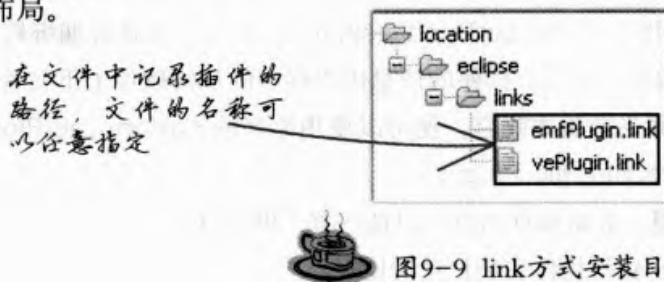


图9-9 link方式安装目录布局

使用link方法的好处是，所有的插件位置被作为一个简单的文本文件来存储。因而可以通过下面代码，

```
path=D:/location/new-plugins/ve
```

简单地将 links 文件夹复制到新的 Eclipse 安装目录中，以升级 Eclipse 并加入原有插件。该方法加入的插件可以通过同方法一相同的方式卸载。

9.2.2 插件的发现和启动

在Eclipse 3.1以后，支持插件的机制是通过OSGi (Open Service Gateway Initiative, 在第1.4节“插件技术和OSGi”中有简要介绍) 框架实现的，从这一点说插件与OSGi捆绑软件 (OSGi Bundle) 相同。插件可以实现其生命周期的启动和停止方面的特殊功能。每个生命周期方法都包括对BundleContext的引用。

★ 注意: ★

在Eclipse 3.0以前，插件是由插件注册表管理的，可以查询它以获得系统中所有插件的插件描述符；而Eclipse 3.2推荐使用BundleContext来达到此目的。平台注册表在3.2中专注于提供有关扩展和扩展点的信息。

为了同以前的版本兼容，Eclipse插件的实现机制非常复杂，讨论Plugin和Bundle的不同超出了本书的范围，有兴趣的读者可以在www.eclipse.org/equinox中了解更多内容。在本书中，插件和捆绑软件可以认为是同一个概念。



当Eclipse第一次启动时，Eclipse运行时遍历plugins文件夹中的目录，扫描每个插件的清单文件信息，并且建立一个内部模型来记录它所找到的每个插件的信息。这时候没有运行任何代码，因而插件并未启动。只有当插件中的start()方法被调用以后，插件才真正启动起来。

★ 注意: ★

如果必须在Eclipse启动时加载和启动插件，需要扩展org.eclipse.ui.startup扩展点，在这个扩展点中要求指定实现org.eclipse.ui.IStartup接口的类，调用其中的earlyStartup()方法。

可以使用“启动和关闭”首选项页面来禁用插件早期启动。



9.2.3 插件信息的获取

每个捆绑软件有一个象征标识符 (symbolic Name)，这个象征标识符便是插件的唯一标识。不论插件有没有启动，都可以使用Platform.getBundle(symbolicName)来获得捆绑软件对象 (org.osgi.framework.Bundle)。当插件启动后，插件的信息由内存中的捆绑软件上下文管理。如果安装了兼容插件，使当前Eclipse版本同先前版本兼容，便可以使用原来的Platform.getPlugin(pluginID)来获得插件实例 (org.eclipse.core.runtime.Plugin)。

Bundle提供了获取插件的信息。获取插件信息可以使用如下的方法。

★bundle.getEntry("/"),——获得插件安装目录的URL。

★`bundle.getHeaders().get(org.osgi.framework.Constants.BUNDLE_VENDOR)`——返回插件的提供者的名称。

★`bundle.getHeaders().get(org.osgi.framework.Constants.BUNDLE_NAME)`——返回这个插件的可显示标签。

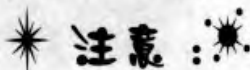
★`bundle.getSymbolicName()`——返回插件的唯一标识。

★`Platform.getResourceBundle(bundle)`——返回当前区域对应的插件资源包对象（`java.util.resourceBundle.`）。这个包对象中的信息将会在插件发布后，作为插件安装目录中的`plugin.properties`文件而存储。

★`new Version((String) bundle.getHeaders().get(org.osgi.framework.Constants.BUNDLE_VERSION))`，——返回插件的版本号。

如果要验证插件是否被激活，则需要检查`bundle`的状态，代码如下所示。

```
bundle.getState == org.osgi.framework.Bundle.ACTIVE
```



在3.1以前的版本中可以使用`IPluginDescriptor`来获取插件的相关信息。由于3.2.1版本中`IPluginDescriptor`类已经不再使用，因而不能再调用它的方法来获取插件的信息。大多数原来的方法都移到了`Platform`中。



9.3 插件的扩展模式

前面说了插件通过扩展模式来实现功能的拓展，那么，这种拓展的具体细节是什么呢？在详细介绍之前，先引入三个概念，宿主插件（host plug-in）、扩展者插件（extender plug-in）和回调对象（call-back object）。宿主插件是提供扩展点的插件，扩展者插件是使用扩展点的插件，它们通过回调对象通信。三者之间的关系如图9-10所示。

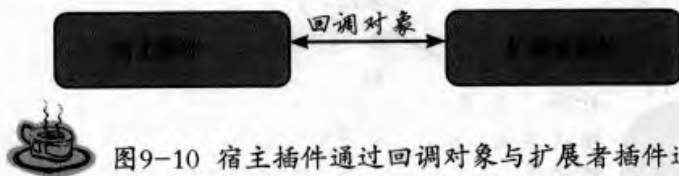


图9-10 宿主插件通过回调对象与扩展者插件通信

9.3.1 扩展和扩展点

如图9-11所示是一个使用Eclipse扩展模式的实例。图中扩展者插件`org.eclipse.search`在宿主插件`org.eclipse.ui`中扩展了一个动作集（`actionSets`）。这是通过扩展UI插件类声明的`actionSets`扩展点来实现的。运行Eclipse，选择“搜索”→“搜索”按钮之后，出现了如图9-12所示的对话框。这是一个非常有创意的思想，在最初构建插件`org.eclipse.ui`时，UI插件并不知道是否存在`org.eclipse.search`插件，但创建完`org.eclipse.search`插件之后，它竟然在不改动UI插件的情况下运行了！只要使用恰当的形式，还可以为Eclipse的界面添加更多的菜单项。这意味着Eclipse插件开发者可以在丝毫不改动原来开发的软件代码的基础上，为其增添新的功能！

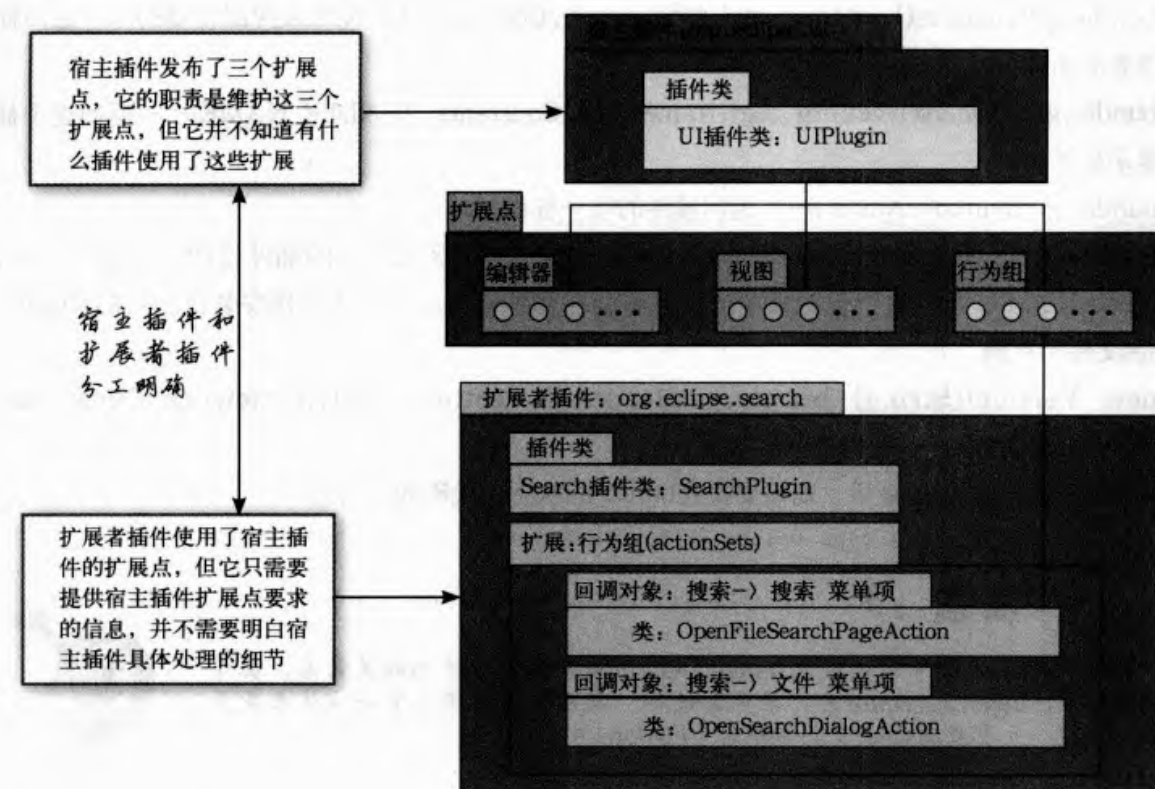


图9-11 插件扩展的参与者

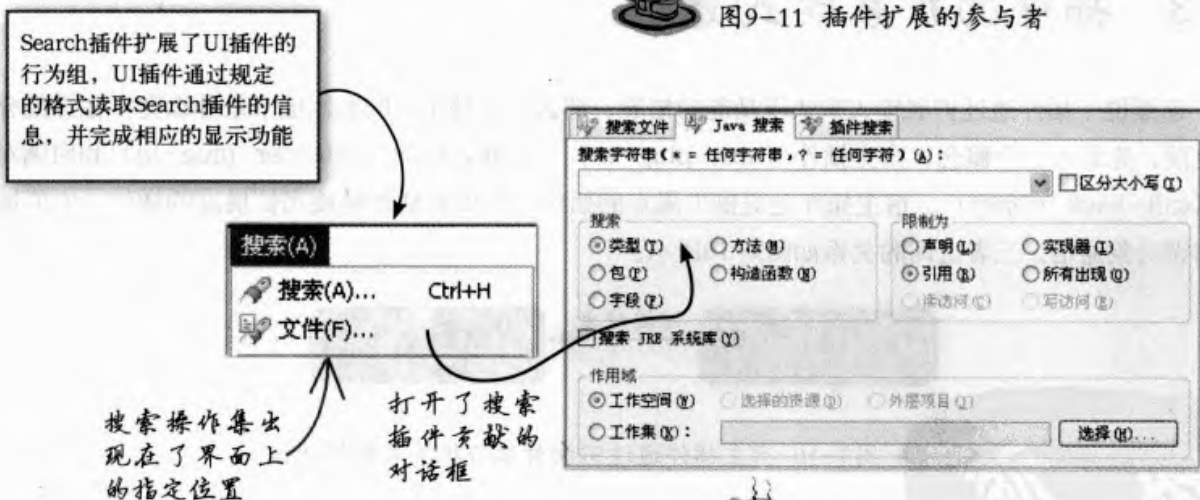


图9-12 搜索对话框

为插件增加一些处理元素被称作对插件的扩展。比较常见的扩展是增加界面元素，插件需要为Eclipse的工作台添加一些界面元素使得插件对用户直接可用。扩展改变了宿主插件的某些行为。改变的行为包括为宿主插件增加处理元素和为这些处理元素定制相应的动作。在图9-11的例子中，org.eclipse.search插件为org.eclipse.ui插件增加了两个菜单项，即“搜索 (Search)” → “搜索 (Search)” 和 “搜索 (Search)” → “文件 (File)”；并且定制了相应的行为，OpenFileSearchPageAction和OpenSearchDialogAction。扩展者插件和宿主插件通过这两个菜单项通信。

在Eclipse中，每个插件都有一个相应的plugin.xml清单文件与其相对应。它们在其中声明了该插件的所有扩展和扩展点。下面两段代码是与图9-11相对应的清单，①处定义了扩展点id：actionSets，这不是该扩展点的完整标识，如果要扩展该扩展点，必须在前面加入其所在插件的id：org.eclipse.ui.actionSets。扩展点的Schema是指扩展点所用的模式，在这里面有对该扩展点的确切定义。

```
<plugin>
  <!-- 工作台扩展点 -->
  <!-- 忽略一些代码 -->
  ①<extension-point id="actionSets" name="%ExtPoint.actionSets"
    schema="schema/actionSets.exsd"/>
  <!-- .....-->
</plugin>
```

声明扩展清单中的每个XML元素和它们的属性都是在Schema中定义的，不同的扩展点这些元素和属性都可能不同，代码如下所示。

```
<!-- 动作集 -->
②<extension point="org.eclipse.ui.actionSets">
  <actionSet
    id="org.eclipse.search.searchActionSet"
    label="%search"
    visible="true">
    <!-- 搜索菜单 -->
    <menu
      id="org.eclipse.search.menu"
      label="%searchMenu.label"
      path="navigate">
      <groupMarker name="internalDialogGroup"/>
      <separator name="fileSearchContextMenuActionsGroup"/>
    </menu>
    <!-- 跳过一些代码 -->
    <!-- 会话组 -->
    <!-- 跳过一些代码 -->
    ③<action id="org.eclipse.search.OpenSearchDialog"
      definitionId="org.eclipse.search.ui.openSearchDialog"
      toolbarPath="Normal/Search"
      menubarPath="org.eclipse.search.menu/internalDialogGroup"
      label="%openSearchDialogAction.label"
      tooltip="%openSearchDialogAction.tooltip"
      icon="$nl$/icons/full/etool16/search.gif"
      helpContextId="open_search_dialog_action_context"
    </action>
    ④
  <class="org.eclipse.search.internal.ui.OpenSearchDialogAction"/>
  <!-- ..... -->
</extension/>
```



在②中可以看到，在声明扩展的时候使用了扩展的完整id。③处定义了一个行为（action），在该行为中声明了OpenSearchDialogAction类（④），该类便是回调对象，回调对象类由扩展者插件定义，当用户第一次使用“搜索”→“搜索”菜单项时，由宿主插件创建实例。

9.3.2 扩展加载过程

加载扩展的全过程如图9-13所示。

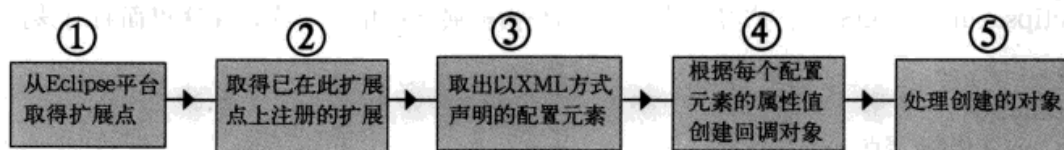


图9-13 加载扩展过程

★Eclipse平台会在启动时读入所有的清单文件，因而启动Eclipse之后，所有需要使用的扩展点都会记录在内存中。Eclipse使用插件注册表来管理插件的所有扩展点和扩展信息，因而可以使用如下的方法来获得扩展点。

```
IExtensionRegistry. getExtensionPoint("org.eclipse.ui.actionSet");
```

★创建扩展对象仍然需要解读清单文件，查找类似<extension point="org.eclipse.ui.actionSets">的扩展声明。要获得全部扩展对象可以使用①中方法获得的扩展点，代码如下所示。

```
extensionPoint.getExtensions();
```

该方法返回记录所有扩展（IExtension对象）的数组。

★在上节中的❸处的action即为一个配置元素。使用②中方法获得扩展数组来获得配置元素，代码如下所示。

```
Extensions[ i ].getConfigurationElements();
```

★创建回调对象的代码如下所示。

```
configurationElements[ j ].createExecutableExtension("class");
```

9.3.3 常用扩展点

1.org.eclipse.ui.views

扩展点 org.eclipse.ui.views 允许插件将视图添加到工作台中。添加视图的插件必须在它们的 plugin.xml 文件中注册该视图，并提供有关该视图的配置信息，例如，它的实现类、它所属视图的类别（或组）以及应该用来在菜单和标签中描述该视图的名称和图标。

2.org.eclipse.ui.editors

插件使用工作台扩展点 org.eclipse.ui.editors 将编辑器添加到工作台中。添加编辑器的插件必须在它们的 plugin.xml 文件中注册编辑器扩展及编辑器的配置信息。

3.org.eclipse.ui.popupMenus

org.eclipse.ui.popupMenus 扩展点允许插件添加到其他视图和编辑器的上下文菜单。可以利用操作的标识将操作添加至特定上下文菜单（viewerContribution），或者通过将操作与特定对象类型

进行关联来添加操作（objectContribution）。

4.org.eclipse.ui.actionSets

插件可以使用 org.eclipse.ui.actionSets 扩展点向工作台菜单和工具栏添加菜单、菜单项和工具栏项。为了减少同时显示每个插件的菜单添加项而导致的混乱，可将添加项分成多个操作集，通过用户首选项来使这些操作集可视。

5.org.eclipse.ui.viewActions

如果插件要为工作台中已经存在的视图添加行为，可以通过org.eclipse.ui.viewActions 扩展点来完成。此扩展点允许插件为现有视图的本地下拉菜单和本地工具栏添加菜单项、子菜单和工具栏条目。

6.org.eclipse.ui.editorActions

当编辑器活动时，org.eclipse.ui.editorActions 扩展点允许向工作台菜单和工具栏作添加条目。

9.3.4 小结

扩展机制在Eclipse中是一个相当普遍的概念，总结一下宿主插件和扩展插件、扩展和扩展点之间的关系对理解扩展机制是很有帮助的，具体总结如下所示。

1. 一个宿主插件可能存在多个扩展点。
2. 一个插件可能既是宿主插件又是扩展插件。
3. 多个插件可能扩展一个扩展点。
4. 一个插件可能多次扩展一个扩展点。
5. 一个扩展插件可能扩展了不同宿主插件的不同扩展。
6. 一个插件的一个扩展可能会被创建多个回调对象。
7. 插件可以扩展它自己定义的扩展点。

9.4 本章小结

本章介绍了Eclipse的平台体系结构以及插件的安装、扩展方式，让读者对Eclipse及其插件体系有一个全局性的了解，为后面创建插件奠定基础。下一章将通过创建一个简单的插件，介绍完整的插件开发过程。



第10章

开发第一个插件项目

拉开崭新的学习帷幕

上一章已经介绍了Eclipse的整体框架，对插件做了比较全面的概述。本章将通过开发一个简单却完全有效的插件项目，以熟悉开发插件的整个过程。在本书后续章节的介绍中，将逐步改进这个插件，最终形成一个功能完善的应用程序。本书中将该插件命名为地址本插件。

本章内容包括：

- ★创建插件工程。
- ★“插件开发”透视图的介绍。
- ★插件的工程结构和详细介绍。
- ★插件程序的运行、调试和发布。



进入第10章

10.1 创建插件工程

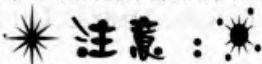
插件在工作空间中被封装在一个工程中，因而创建插件的第一步是要创建一个插件工程。Eclipse 为开发者提供了“新建插件项目”向导，这个向导除了用来创建插件工程和插件所必须的清单文件、属性文件之外，还提供了多种多样的代码产生选项，用来为具有不同侧重点的插件创建示例插件代码。开发者可以借助这些选项快速进行特定目的的插件开发。本章将使用其中的视图代码模板，在其自动生成的代码上构建地址本插件，从而使得地址本插件的构建过程简单快速。

10.1.1 新建插件

首先，运行Eclipse平台，从主菜单中选择“文件（File）”→“新建”→“项目...”，并产生的“新建项目”对话框中选择“插件项目”，打开“新建插件项目”向导，如图10-1和图10-2所示。

在“新建插件项目”向导中，首先需要定义项目的结构。在“创建插件项目”页，输入项目的名称。项目名称可以为任何形式，但是不能以点号（.）结尾，通常遵循逆向命名约定（如org.eclipse.ui）。本例项目名称为com.plugindev.addressbook。

在“目标平台”区域中选择插件运行的框架，这个选项控制向导后面“插件内容”页上的代码生成选项以及可用的模板列表。在这个选项中，Eclipse框架是指使用了Eclipse扩展注册表（IExtensionRegistry，见第9章）的插件。IExtensionRegistry是在Eclipse平台的运行时层提供的，因而大部分的Eclipse插件都会使用此框架。如果不希望依赖Eclipse平台的运行时层，就要使用OSGi框架。本书中涉及的所有插件，均使用Eclipse框架。



在OSGi框架中，又可以在Equinox和标准中选择。它们的区别在插件的Manifest.mf文件（见本章10.4.2节）中。Equinox OSGi框架使用了一些特定于Eclipse的头及属性。如果不希望使用这些扩充，可以选择“标准”。



在项目设置区域中选择“Java项目”，表示要为插件添加Java代码。有些插件不需要添加Java代码，则需要去掉该选项前的复选框。在Java项目中，需要指定默认的原代码文件夹和输出文件夹，编译过的二进制.class文件将会存放在输出文件夹中。保留所有默认值，单击“下一步”按钮。

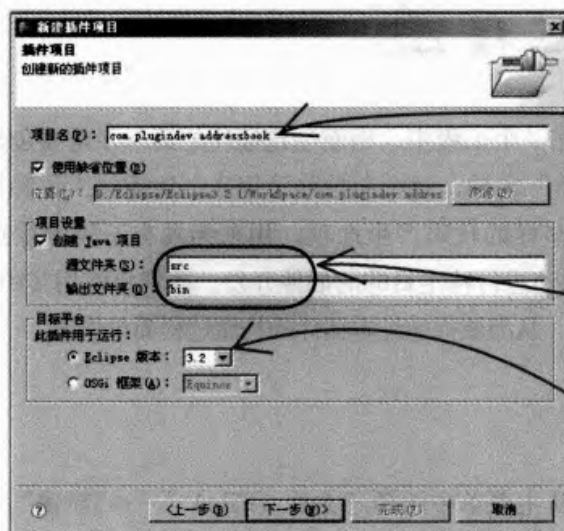
可以在此过滤树中的内容



选择插件项目，
为创建插件工程
做好准备



图10-1 “新建项目”对话框



设置项目的名称
为 `com.plugindev.
addressbook`

设置默认的源代码文件
夹和二进制代码输出文
件夹，以后可以在插件
开发环境中修改

选择Eclipse的版本



图10-2 “新建插件项目”向导的“插件项目”页

10.1.2 使用向导

向导的下一页是关于插件的内容信息，该页用来定制插件数据，如图10-3所示。在插件属性区域中，插件的标识，版本号，插件的名称都是必填的。插件标识默认情况下和插件名称保持一致。“类路径”是指将插件打包后代码所在的位置，通常将其留空。

控制插件生命周期的插件类为激活器（Activator），该激活器会根据是否选择“此插件将对UI进行添加”复选框而继承自不同的类。

如果选择该选框，则激活器类（插件类）将扩展 `org.eclipse.ui.plugin.AbstractUIPlugin` 类来对插件项目的生命周期进行管理。

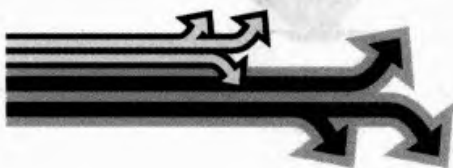
★如果该插件不对UI进行添加，则激活器类扩展 `org.eclipse.core.runtime.Plugin` 类来对插件项目的生命周期进行管理。

★如果在图10-2所示的“插件项目”页选择OSGi框架，激活器则会实现 `org.osgi.framework.BundleActivator` 接口。无论生成的激活器代码是什么，激活器的作用都是基本相同的，它用来控制插件的生命周期。

稍后将要为地址本插件贡献一个视图，因而选择“此插件将对UI进行添加”。

下面的区域“富客户机应用程序”用来创建富客户机应用程序。如果选择“是”，则在“下一步”的“模板”页中，会出现富客户机应用程序的模板。本例先不创建此应用程序。

如果不使用模板，就可以选择“完成”了。本例中选择“下一步”。





在“模板”页中，出现了各种插件片段模板，如图10-4所示。这些模板是入门，以及快速开发非常好的帮手，可以使用它们自动产生相应的代码。本例中，选择“具有视图的插件”，单击“下一步”按钮。

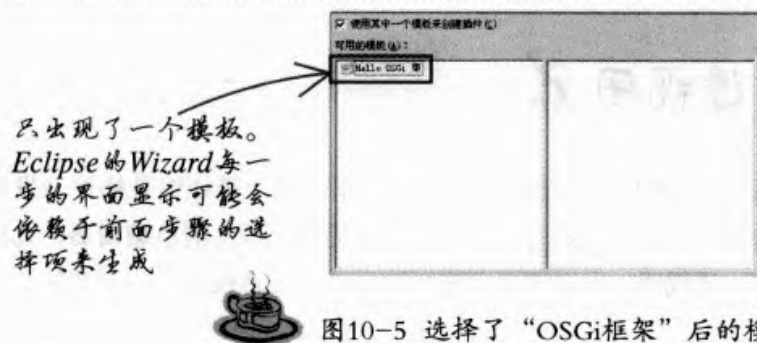
※ 注意 ※

用户也可以在向导中添加自定义模板，如果要这样做，需要扩展org.eclipse.pde.ui.pluginContent 扩展点。该扩展点用来为新的插件内容提供向导，插入到此扩展点的向导需要实现org.eclipse.pde.ui.IPluginContentWizard 接口且应扩展 org.eclipse.jface.wizard.Wizard。



※ 注意 ※

模板页面由于用户在前面选择的不同而不同。如图10-5所示是在“插件项目”页选择“OSGi框架”后的模板页，在此页中，只有“Hello OSGi”一个模板。



选择“下一步”后，进入“主视图设置”页面，如图10-6所示。此页面，中提供了把视图添加进插件所必须的各种信息。去掉“将视图添加到资源透视图”前面的对勾，以方便今后自定义透视图。选择“下一步”。在“视图功能部件”页面中，如图10-7所示保留默认值，以便将功能部件全部选中，单击“完成”按钮。

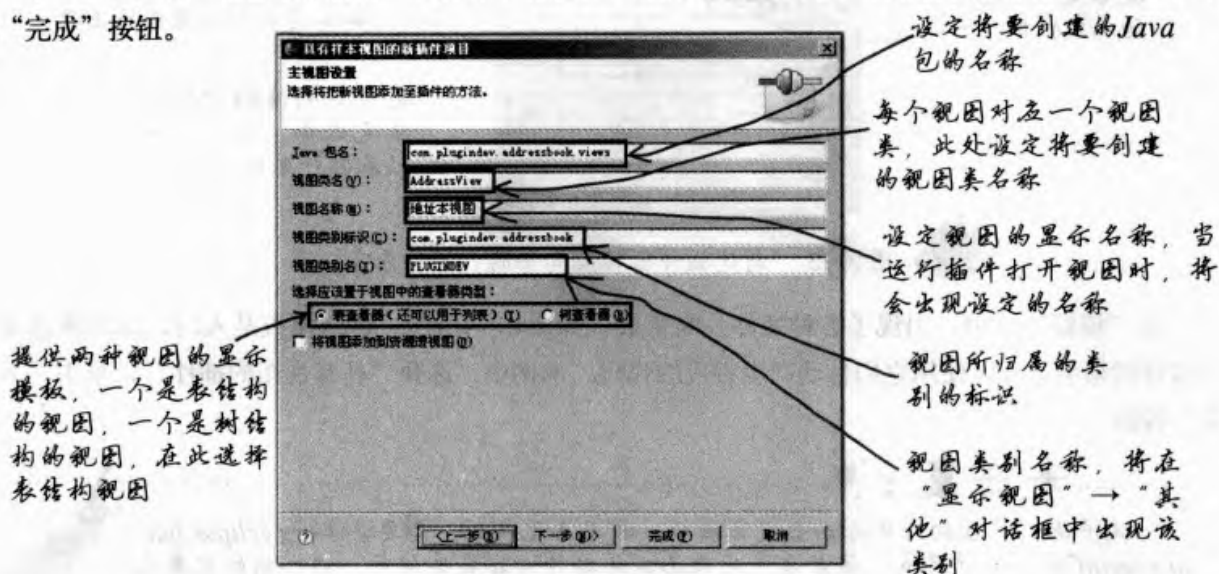
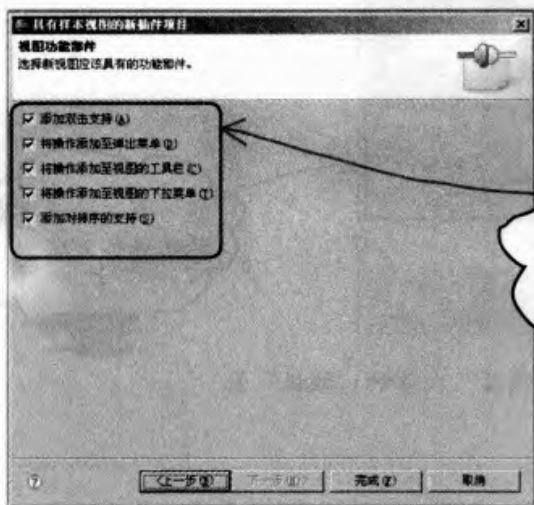


图10-6 “具有样本视图的新插件项目”的“主视图设置”页



添加视图应该添加的功能。为了产生较为全面的视图的编程框架，将所有模板支持的功能全都选中



图10-7 “具有样本视图的新插件项目”的“视图功能部件”页

10.2 “插件开发”透视图

上一节的操作在选择“完成”之后，就进入了“插件开发”透视图。这个透视图是插件开发环境提供的。插件开发环境为查看插件的各个方面提供了几种视图，并且提供了插件清单编辑器，为编辑插件的清单提供了方便。要手动打开该透视图，可以选择“窗口”→“打开透视图”→“其他”→“插件开发”。

10.2.1 PDE视图

选择“窗口”→“打开视图”→“其他”，展开其中的“PDE”类别，可以看到“插件”视图和“插件依赖项”视图两个视图。

1. “插件”视图

“插件”视图用来显示正在工作空间中开发的所有插件以及构成目标平台的插件树列表，可以在这里快速查看已存在的插件，如图10-8所示。双击每个插件，就会在工作空间中打开它的清单文件。

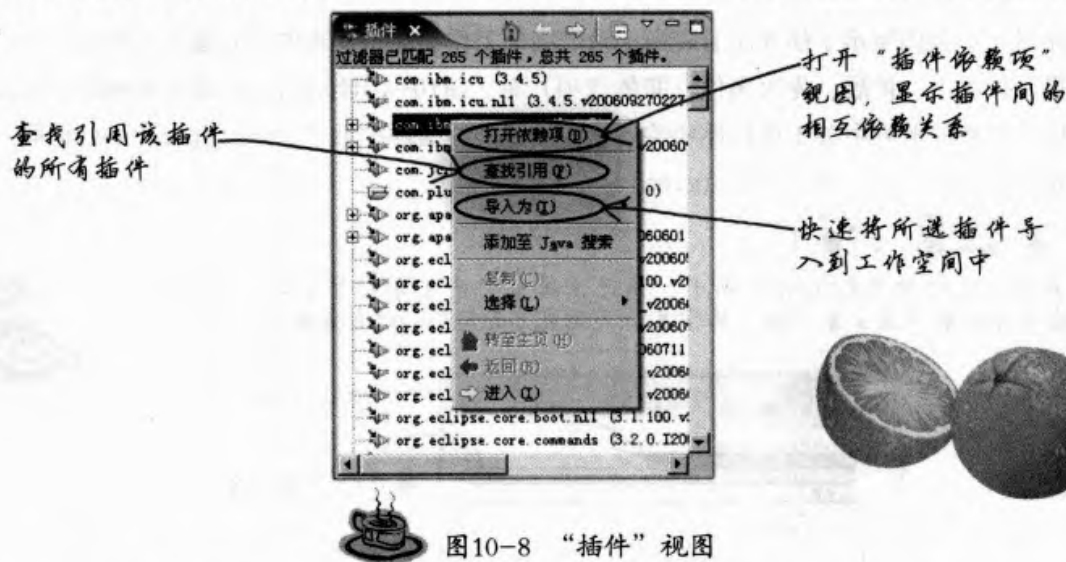


图10-8 “插件”视图

选中某个插件后，单击右键，弹出上下文菜单。此菜单中的“打开依赖项”、“查找引用”和“导入为”操作很有用。图10-8中阐述了这几个操作的功能。

2. “插件依赖项”视图

“插件依赖项”视图显示了插件间的相互依赖关系，此视图需要聚焦到某个插件上。当视图打开时，单击右键，选择“聚焦于”，在弹出对话框中选择第10.1节创建的插件com.plugindev.addressbook（地址本插件），如图10-9所示。由图中可知，该插件直接依赖于org.eclipse.core.runtime和org.eclipse.ui两个插件。

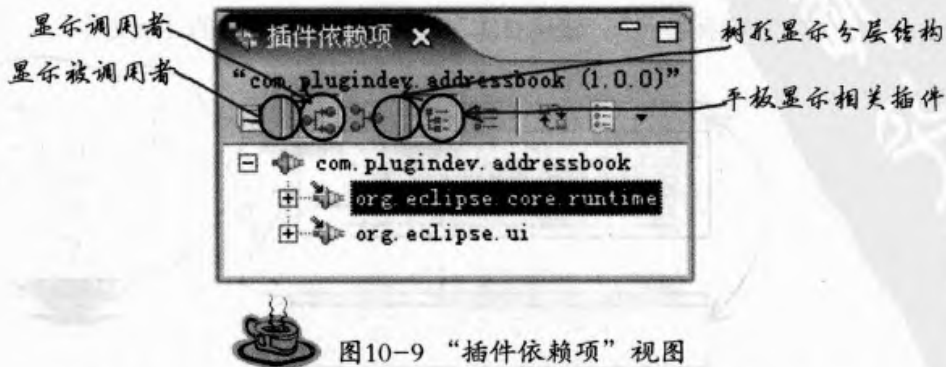


图10-9 “插件依赖项”视图

插件依赖项视图中提供了分层次显示和平板显示两种显示方式，并将插件分为调用者和被调用者

两种角色，可以在视图的工具栏中在不同的显示方式中切换。

10.2.2 PDE运行时视图

在“窗口”→“打开视图”→“其他”的“PDE运行时”分类中，也包含“插件注册表”视图和“错误日志”视图两个视图。

1. “插件注册表”视图

“插件注册表”视图显示了插件注册表中记录的当前工作空间中发现的所有插件的相关信息，如插件所在位置，扩展点、扩展、先决条件（即依赖项）等。图10-10显示的是启动包含地址本插件的Eclipse实例后，插件注册表中关于该插件相关信息的描述。该插件扩展了org.eclipse.views扩展点，先决条件为org.eclipse.core.runtime和org.eclipse.ui。

※ 注意：※

在开发地址本插件的Eclipse实例中，插件注册表中并没有关于地址本插件的信息，要在插件注册表中显示某个插件的信息，必须在启动Eclipse时包含该插件。

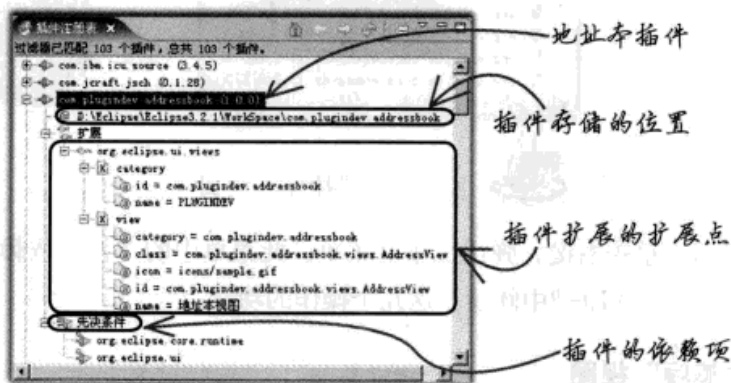


图10-10 “插件注册表”视图

2. “错误日志”视图

“错误日志”视图记录插件的错误和警告信息。这些信息存储在工作空间中的.metadata目录下的.log文件中。如图10-11所示是一个“错误日志”视图的实例和对应的.log文件。

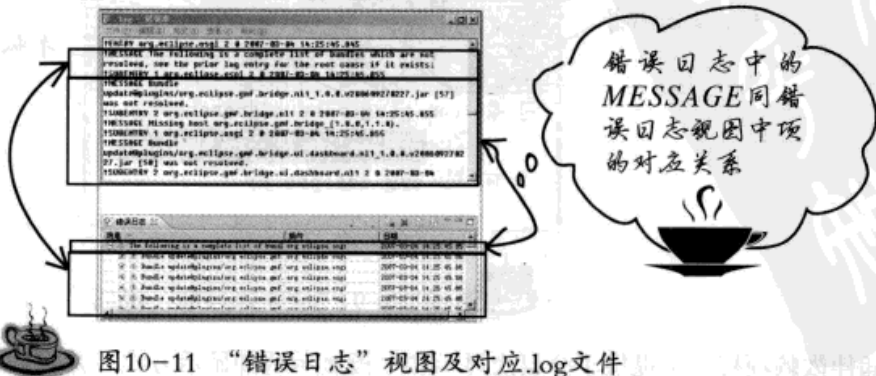


图10-11 “错误日志”视图及对应.log文件

10.2.3 清单编辑器

PDE 提供了一个基于表单的多页插件清单编辑器。打开地址本插件，可以看到，清单编辑器包括“概述”页、“依赖性”页、“运行时”页、“扩展”页、“扩展点”页、“构建”页和 MANIFEST.MF、plugin.xml、build.properties 9个页面。它们为编辑插件的各种清单文件提供了极大的方便。前6个页面是可视化操作页面，后三个页面是特定的文本编辑器。图10-12显示了插件清单编辑器的“概述”页面。

“概述”页面是快速进入其他各个页面的通道，在这个页面中，有指向其他各页面的链接。该页面还提供了对插件的一般信息和执行环境的编辑，插件开发者可以在这里定义和修改插件的基本属性（修改在创建插件工程时对插件做出的默认设定）。

接下来的第10.3节将通过介绍地址本插件的结构，结合插件清单的格式来介绍清单编辑器的其他各个页面。

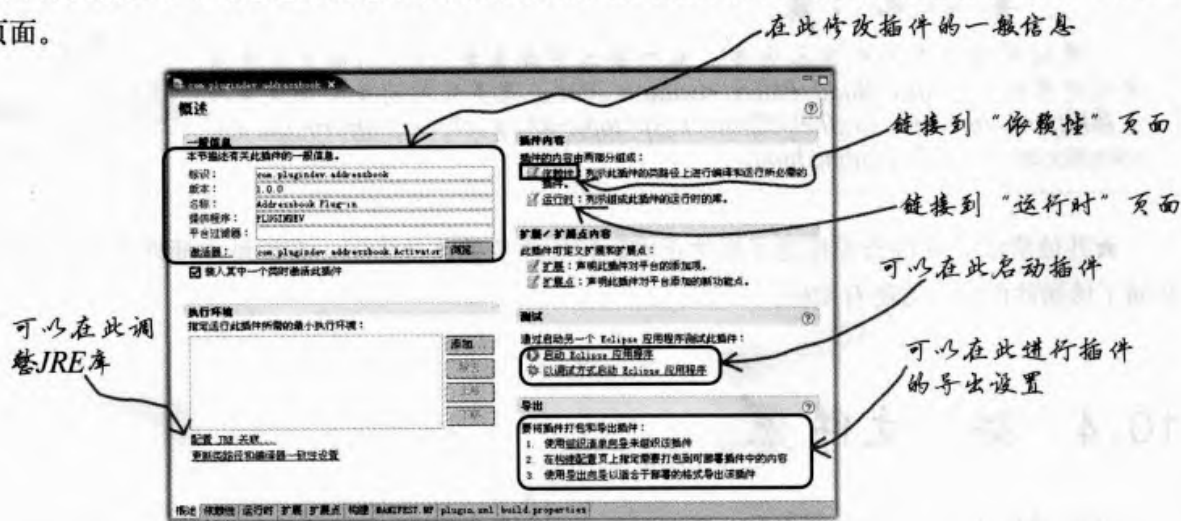


图10-12 插件清单编辑器的“概述”页面

10.3 插件工程结构

图10-13显示了工作空间中的地址本插件包括的4个部分。



图10-13 地址本插件工程的结构



★源代码部分。源代码部分包括了模板自动创建的两个包，其中，`com.plugindev.addressbook`包中包含了同插件整体相关的类，激活器就包含在这个包中；`com.plugindev.addressbook.views`包中包含了同视图相关的类，里面有通过模板创建的地址本视图（`addressview`）类。

★插件文件。插件文件是构成一个完整插件至关重要的部分。在Eclipse3.2中，插件文件包括以下三个子部分。

◆插件清单文件`plugin.xml`。

◆OSGi捆绑软件清单文件`MANIFEST.MF`。

◆`build.properties`文件。

◆所依赖的系统库。系统库包括两种，一种是JRE系统库，可以在插件清单编辑器的“执行环境”中修改；另一种是插件的依赖项，也就是插件运行所需要的其他插件。

※ 注意 : ※

有时可能需要装入第三方库。装入第三方库需要在OSGi捆绑软件清单文件中加入Eclipse-BuddyPolicy: <value>头部。更多信息请参阅在线帮助文档http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/misc/buddy_loading.html。





★其他资源。其他资源描述了插件正确运行所需要的其他资源。在地址本插件中，`icons`文件夹存储了该插件所需要的所有图标。

10.4 插件文件

PDE 的插件清单编辑器能够统一管理所有插件文件，即`MANIFEST.MF`、`plugin.xml`和 `build.properties`（有的插件还会有`fragment.xml`）。使用插件清单编辑器时，就像是在编辑单个文件一样；但 PDE 会执行将数据写入磁盘上不同文件这一任务。PDE的清单编辑器也提供了文本编辑页面来支持直接文本编辑。本节将展示文本编辑页面和表单页面的一一对应关系，并以此为基础，介绍插件文件的格式。

10.4.1 plugin.xml文件

由于Eclipse体系结构基于可配置扩展点这一概念，因而插件具有扩展其他插件和发布扩展点的能力。`plugin.xml`文件便是用来记录插件的扩展点和扩展的。在地址本插件中，暂时还没有扩展点，仅有一个对Eclipse视图的扩展。打开清单编辑器中的`plugin.xml`页面，如图10-14中的  所示，可以在这里手动编辑此XML文件。这个文件对应着扩展页面，如图10-14中的  所示，在这里采用表格化的方式编辑同在XML文件中编辑效果是相同的，改变这两处的任何一处，都会改变`plugin.xml`文件。

可以通过图10-14大致了解一个`plugin.xml`文件的结构，具体如下所示。

★<plugin> 元素定义清单的主体。它可以有选择地包含由该插件引入的任何新扩展点的声明，

以及功能扩展的配置。

★<extension> 定义该插件对其他插件的功能扩展。它具有point（所引用的扩展点的标识）、id（此扩展实例的标识）、name（扩展提供给用户的名称），如图10-14的“扩展详细信息”域所示。对于不同的属性元素，有不同的子属性，如图10-14中的所示。

★还有一个重要部分便是扩展点，由<extension-point>元素定义。图10-14中未给出。

※ 注意 :

扩展点和扩展的标签在plugin.xml中很容易混淆。扩展点以<extension-point>标签开头，而扩展以<extension>标签开头，但其属性元素point往往紧跟其后。注意区分这两个标签。

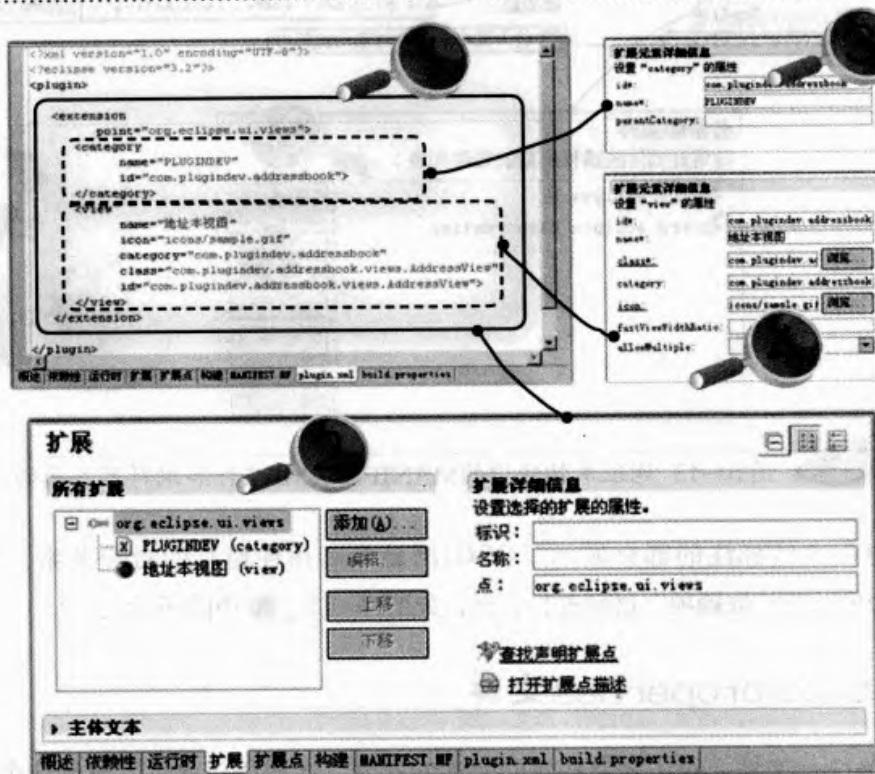


图10-14 插件清单文件及其对应编辑器扩展页

10.4.2 MANIFEST.MF文件

MANIFEST.MF文件用来提供关于捆绑软件的描述信息。地址本插件中的MANIFEST.MF文件如下所示。

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Addressbook Plug-in
Bundle-SymbolicName: com.plugindev.addressbook; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: com.plugindev.addressbook.Activator
Bundle-Vendor: PLUGINDEV
```




```
Bundle-Localization: plugin
Require-Bundle: org.eclipse.ui,
org.eclipse.core.runtime
```

MANIFEST.MF文件同plugin.xml文件类似,同样可以通过两种方式编辑。如图10-15所示,“概述”页面^①和“依赖项”页面^②均同它相关联。

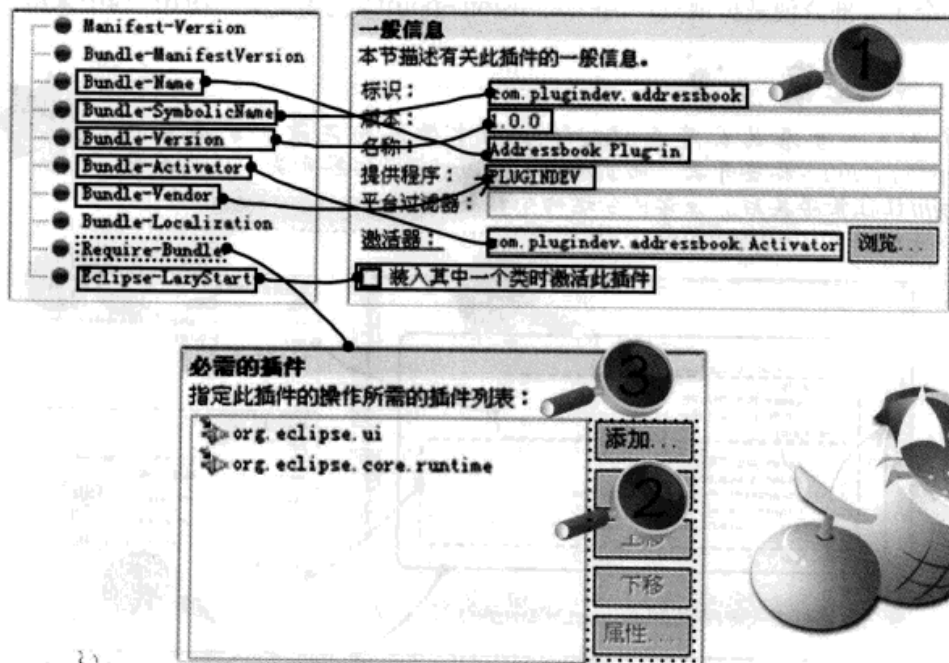


图10-15 地址本插件中的MANIFEST.MF文件和编辑器页面的关系

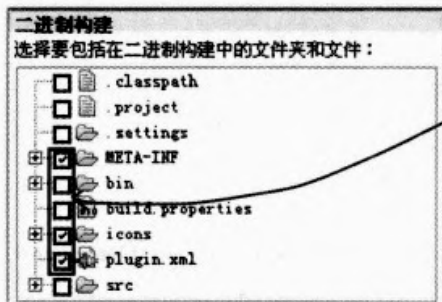
图10-15中用实线标注的部分显示了OSGi清单头同插件信息的对应关系。用虚线标注的Require-Bundle可以在“依赖项”页面进行管理,如图10-15^②中的所示。

10.4.3 build.properties文件

build.properties文件记录了需要构建的元素的列表,这样PDE就可以通过这个文件来构建插件,并且在最终构建的插件中不需要再包含这个文件。build.properties文件如下所示,其中包括源目录、输出目录和bin文件夹所包括的目录。

```
Manifest-Version: 1.0
source.. = src/
output.. = bin/
bin.includes = plugin.xml,\
               META-INF/, \
               ., \
               icons/
```

在清单编辑器的“构建”页面中,可以编辑这些选项。图10-16显示了地址本中在bin.includes里默认包括的资源。



打对勾的项表示二进制目录中包括的资源。如果插件依赖于额外的资源，一定要包含在构建路径中



图10-16 地址本中二进制构建所包括的项

10.5 插件类

插件类在前面已经介绍很多了，下面为插件类的代码。

```
package com.plugindev.addressbook;

import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.plugin.AbstractUIPlugin;
import org.osgi.framework.BundleContext;

/**
 * The activator class controls the plug-in life cycle
 */
public class Activator extends AbstractUIPlugin {

    // The plug-in ID
    public static final String PLUGIN_ID = "com.plugindev.addressbook";

    // The shared instance
    private static Activator plugin;

    /**
     * The constructor
     */
    public Activator() {
        plugin = this;
    }

    public void start(BundleContext context) throws Exception {
        super.start(context);
    }

    public void stop(BundleContext context) throws Exception {
        plugin = null;
        super.stop(context);
    }

    /**
```

启动插件时调用该方法

卸载插件时调用该方法





```


    * Returns the shared instance
    */
    public static Activator getDefault() {
        return plugin;
    }

    /**
     * Returns an image descriptor for the image file at the given
     * plug-in relative path
     */
    public static ImageDescriptor getImageDescriptor(String path) {
        return imageDescriptorFromPlugin(PLUGIN_ID, path);
    }
}

```

获得插件中图像的
描述符

一般情况下，插件类声明一个静态字段，用于引用这个唯一的实例，如  处所示，这就是前面所说的插件标识。 处和  处的start()和stop()方法分别管理插件实例的启动和关闭。

 处是一个很重要的静态方法，此方法可以获得插件中的图像标识符(ImageDescriptor)，依据此标识符可以使用图像资源。例如要在地址本查检中使用此插件中icons目录下的sample.gif图标，可以通过如下方法。

★AbstractUIPlugin.getImageDescriptor(“icons/sample.gif”).createImage();

★插件类还提供了一种获得该插件实例的方法AbstractUIPlugin.getDefault();

10.6 运行插件程序

在工具栏中单击“运行”菜单，启动“创建、管理和运行配置选项”对话框，如图10-17所示。

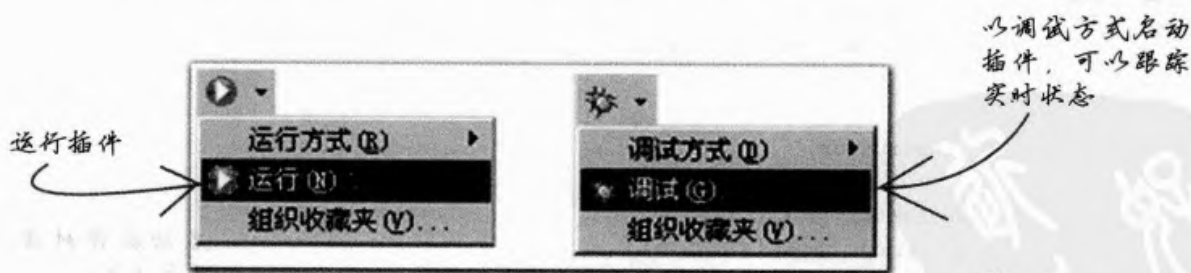


图 10-17 “运行”菜单和调试菜单



在打开的配置对话框中，双击左侧视图“Eclipse应用程序”，新建一个配置，起名为“AddressBook”，单击“应用”按钮，如图10-18所示，然后单击页面右下角“应用”按钮，该对话框关闭，同时新的Eclipse实例将会启动。

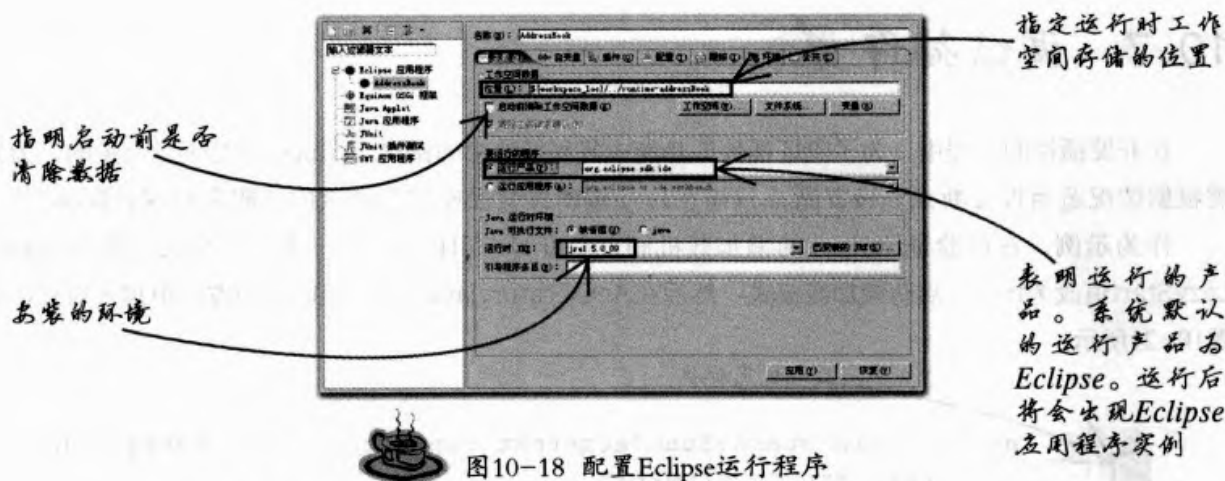


图10-18 配置Eclipse运行程序

※ 注意: ※

在新的Eclipse实例中, 包括了工作空间中和的所有插件。要定制新的Eclipse实例中加载的插件, 可以在“创建、管理和运行配置选项”对话框中的“配置”页面进行管理。



在新的Eclipse实例中, 选择“窗口”→“显示视图”→“其他”, 弹出如图10-19所示的对话框, 可以发现, 刚创建的地址本视图出现在其中了!

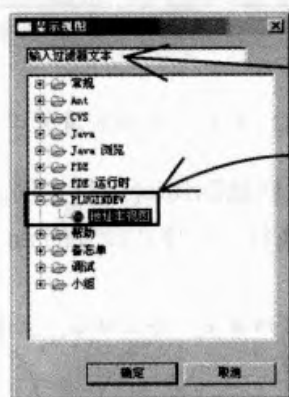


图 10-19 新实例中的“显示视图”对话框

选择“地址本视图”, 单击确定, 地址本视图便被添加进工作透视图。当前的地址本视图功能很简单, 视图中只包括三个元素和两个工具栏菜单按钮, 如图10-20所示。

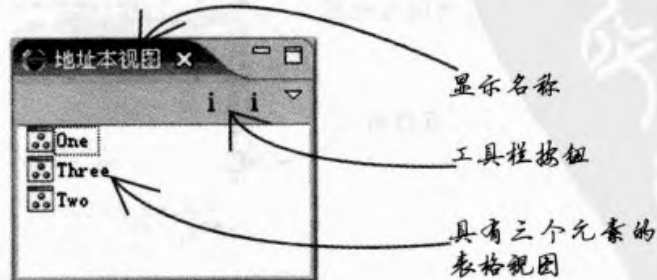


图10-20 地址本视图

10.7 调试插件

在开发插件的过程中，为了验证插件的功能或修复插件中的错误，调试插件是不可避免的。为此需根据情况适当设置断点。设置断点只需在Java编辑器页面中需要调试的代码左侧双击鼠标。

作为示例，在此验证Eclipse的懒加载机制。打开MANIFEST.MF文件，将其中的Eclipse-LazyStart值改为true，启动懒加载模式；然后在Activator.java代码中的start()方法中加入断点，如图10-21所示。

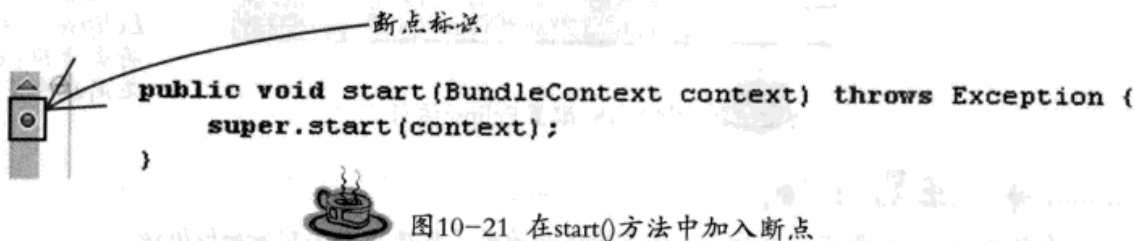


图10-21 在start()方法中加入断点

在使用调试菜单打开配置对话框，将“启动前清除工作空间数据”前面的复选框选中，如图10-22所示。

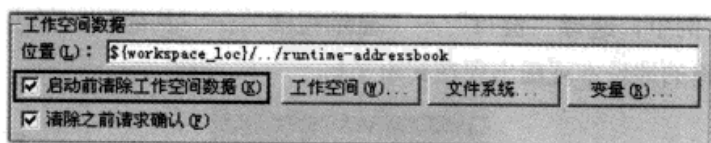


图10-22 选中“启动前清除工作空间数据”

单击“运行”按钮，以调试的方式新建Eclipse运行实例。重复第10.6节的步骤打开“地址本视图”。直到这时start()方法才第一次执行，程序在设置的start()中的断点处停止，如图10-23和图10-24所示。

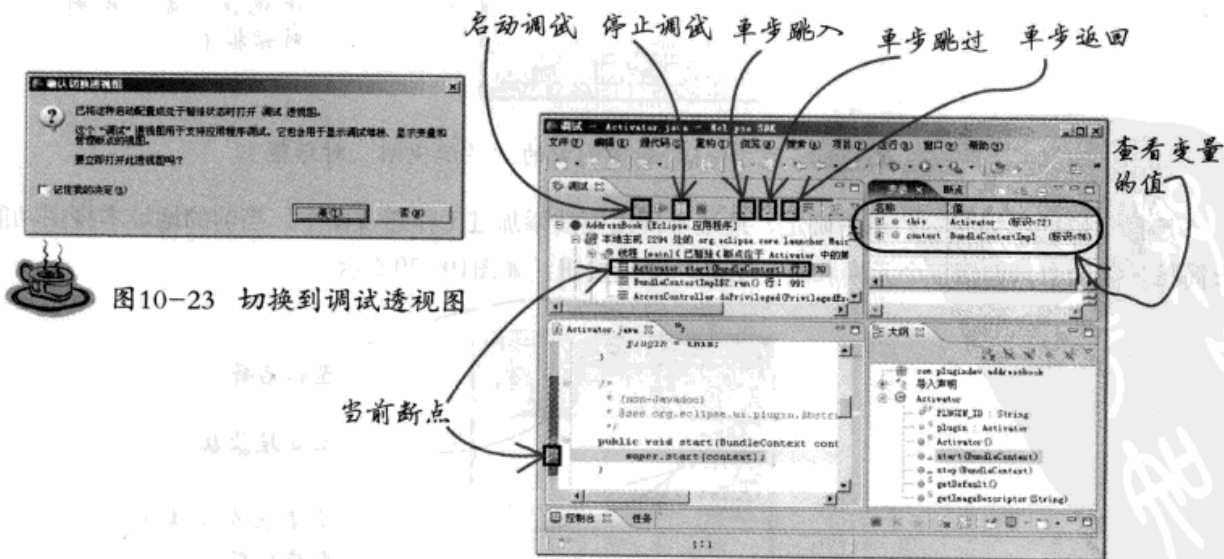


图10-23 切换到调试透视图

图10-24 调试透视图

在图10-24中，可以看到Eclipse强大的调试功能，为编写出健壮的插件提供了保证。在“变量”视图中，可以查看当前变量的值。如果要自定义监视的变量，在“窗口”→“显示视图”中选择“表达式”，在出现的视图的右键菜单中选择“添加查看表达式”，便可以添加想要查看的表达式，如图10-25所示。

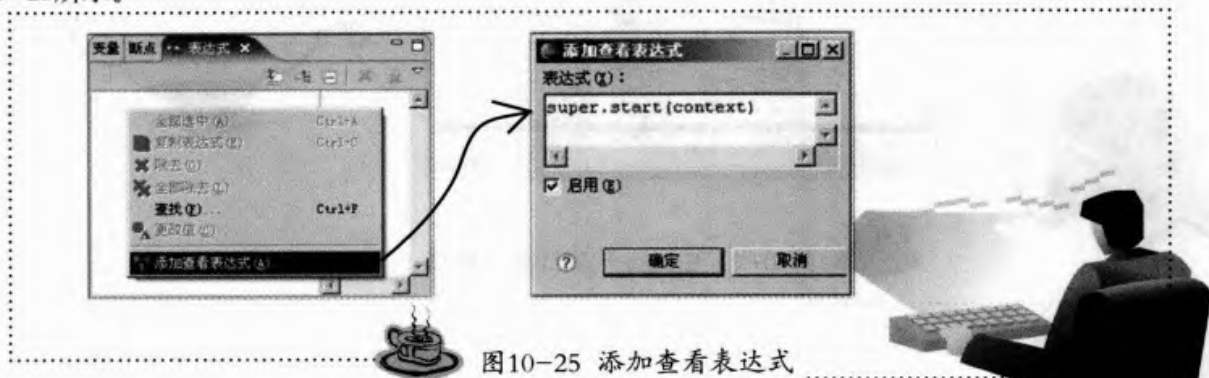


图10-25 添加查看表达式

10.8 发布插件

Eclipse可以有两种方式发布插件，一种需要使用Ant脚本，另一种则只需要使用PDE所提供的“导出”向导便可。选择“文件”→“导出”→“插件开发”→“可部署的插件和段”打开插件导出向导。在其中选择地址本插件，输入想要导出的本地目录，单击“完成”按钮，工作空间中的地址本插件就成为Eclipse产品中可部署的格式存在选定的目录中了。如图10-26所示。



图10-26 导出插件

将产生的.jar文件放入Eclipse程序的plugin文件夹中，重启Eclipse程序，地址本插件就包含在其中了（也可以使用第9.2.1节所述的其他方法来部署插件）。可以通过“帮助”→“Eclipse SDK”，在弹出的对话框中单击“插件详细信息”按钮来查看插件信息。从图10-27所示，前面创建的地址本插件已经包含在其中了，这说明Eclipse平台已经认出了地址本插件。这时就可以在当前的Eclipse应用程序中打开地址本视图了。

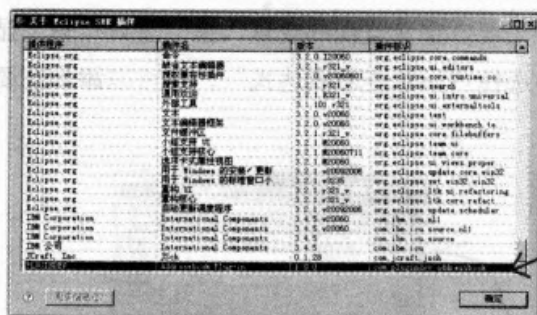
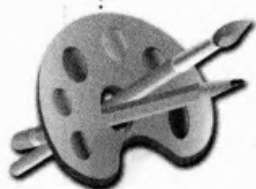



图10-27 Eclipse产品信息对话框

10.9 本章小结

本章通过一个简单的例子，介绍了插件开发环境和创建、运行、调试、发布插件的完整过程。随后的章节将通过不断完善地址本插件示例，为插件开发作更为详细的介绍。



完成工作了，可以逛街去了！哈哈！

第11章 操作 (Actions)

上一章介绍了完整的开发Eclipse插件的过程，从本章开始，将深入Eclipse插件开发的细节，详细介绍插件开发的各个部分。本章将介绍如何为插件贡献操作，这个概念表示一种抽象的用户操作，如打开文件，查找字符串，或者跳转到某个标记等。操作是完成与用户交互的关键，可以按照用户的选择启用相应的功能。

本章内容包括：

- ★概览Eclipse中的各个操作。
- ★使用模板创建操作扩展。
- ★介绍Eclipse的操作代理机制。
- ★为插件添加工作台的各种操作。
- ★为操作关联命令，实现快捷键映射。

拉开崭新的学习帷幕

进入第11章

11.1 Eclipse中的操作概览

在Eclipse IDE中,操作可以在许多地方出现,如菜单栏、工具栏和上下文菜单,这些操作对丰富编辑器的功能,增强同用户的交互能力起到了重要作用。

操作可以在Eclipse IDE中出现的所有位置如图11-1所示。

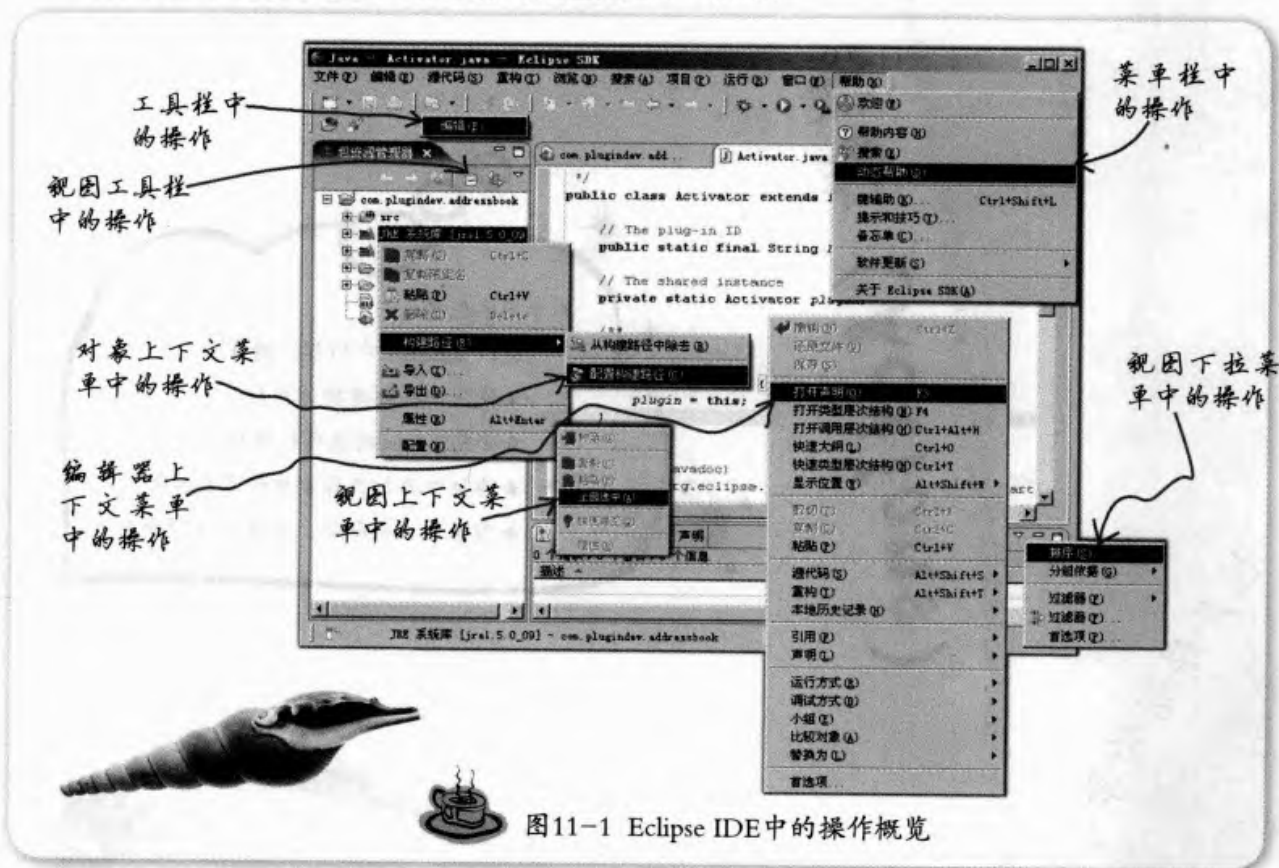


图 11-1 Eclipse IDE 中的操作概览

Eclipse中的操作既可以通过扩展点定义,也可以通过代码定义。本章主要介绍如何使用扩展点机制为Eclipse插件实现各种操作。下面先为Eclipse的工具栏添加一个打开地址本的操作。

11.2 添加工作台窗口操作

在本节中,将对Eclipse的工作台窗口添加“打开视图”的操作。

仔细观察一下Eclipse的工具栏就会发现,工具栏中的按钮被分为多个组,每个组叫一个操作集(action set),用来集合一组相关的操作。如图11-2所示,用于创建Java元素的几个按钮被放在了同一个操作集中。

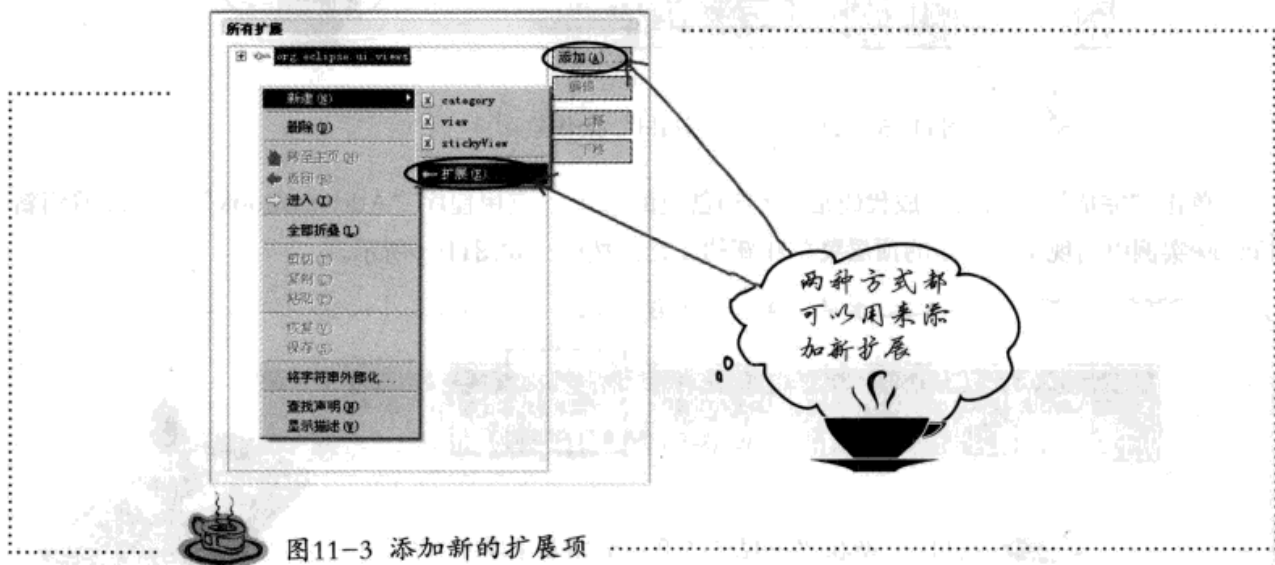


图 11-2 调试和运行操作集

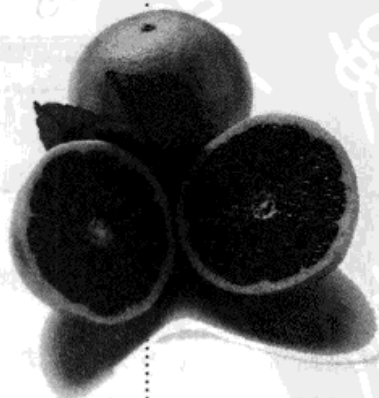
要创建一个新的操作集，让它出现在工具栏中，就必须在“地址本”插件清单中添加一个 actionSet 扩展，用于描述新的操作。

11.2.1 使用模板创建扩展

打开“地址本”插件清单编辑器（双击树视图的 plugin.xml 文件），选择“扩展”页，在“扩展”选项卡中单击“新建”按钮；或者在“所有扩展”区域内单击右键，在弹出的上下文菜单中选择“新建”→“扩展”命令来打开“新建扩展”向导，如图 11-3 所示。



在“新建扩展”向导中，去掉“只显示必需插件中的扩展点”复选框，然后从所有可选择的扩展点中选择 org.eclipse.ui.actionSets，如图 11-4 所示。选中这个扩展点后，在此扩展向导的下部，会出现可以参考的模板“Hello, World”操作集。首先在该模板的指导下创建第一个操作实例。选中该模板，单击“下一步”按钮。



在下一个向导页中，设置Java包名、操作类名和消息框文本，如图11-5所示。

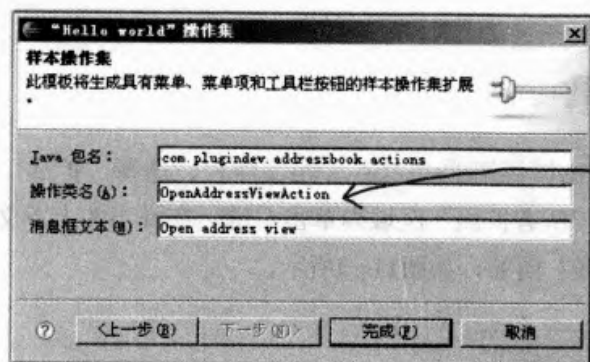


图11-5 生成“Hello world”操作集向导

单击“完成”按钮，生成代码后，启动创建的Eclipse应用程序“AddressBook”，可以看到新Eclipse实例中出现了一个新的顶层菜单和新的工具栏按钮，如图11-6所示。

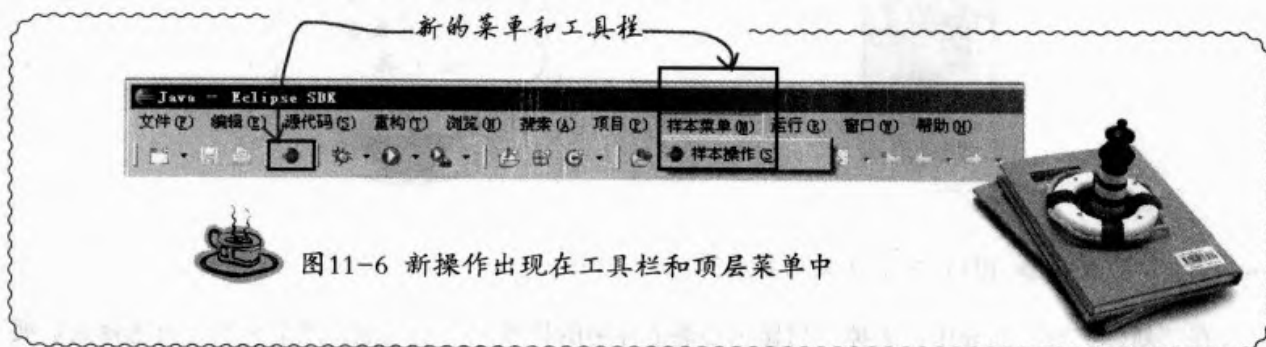


图11-6 新操作出现在工具栏和顶层菜单中

打开该菜单后，弹出一个对话框，对话框的标题是贡献的插件名称，信息是前面输入的信息。如图11-7所示。

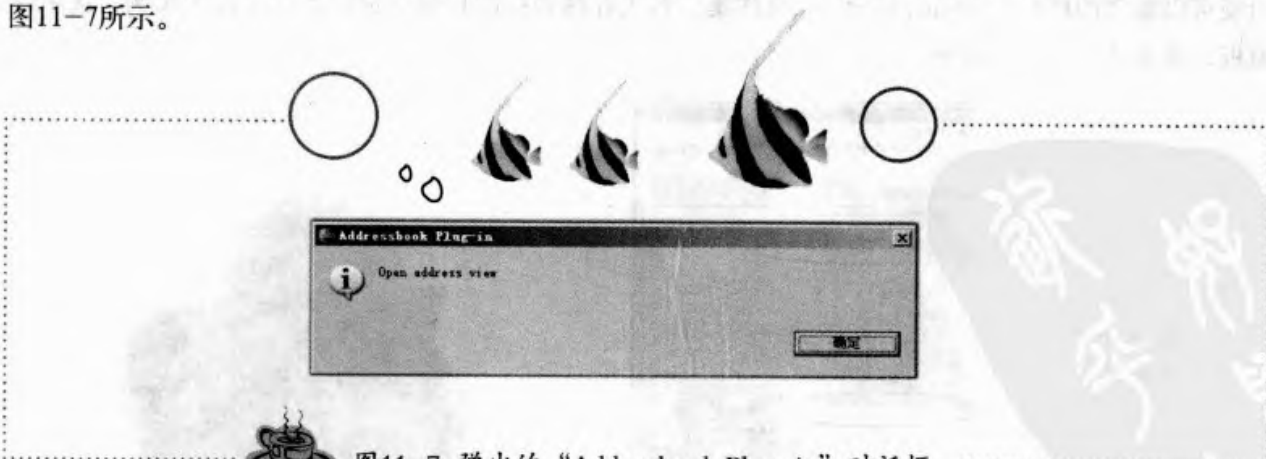
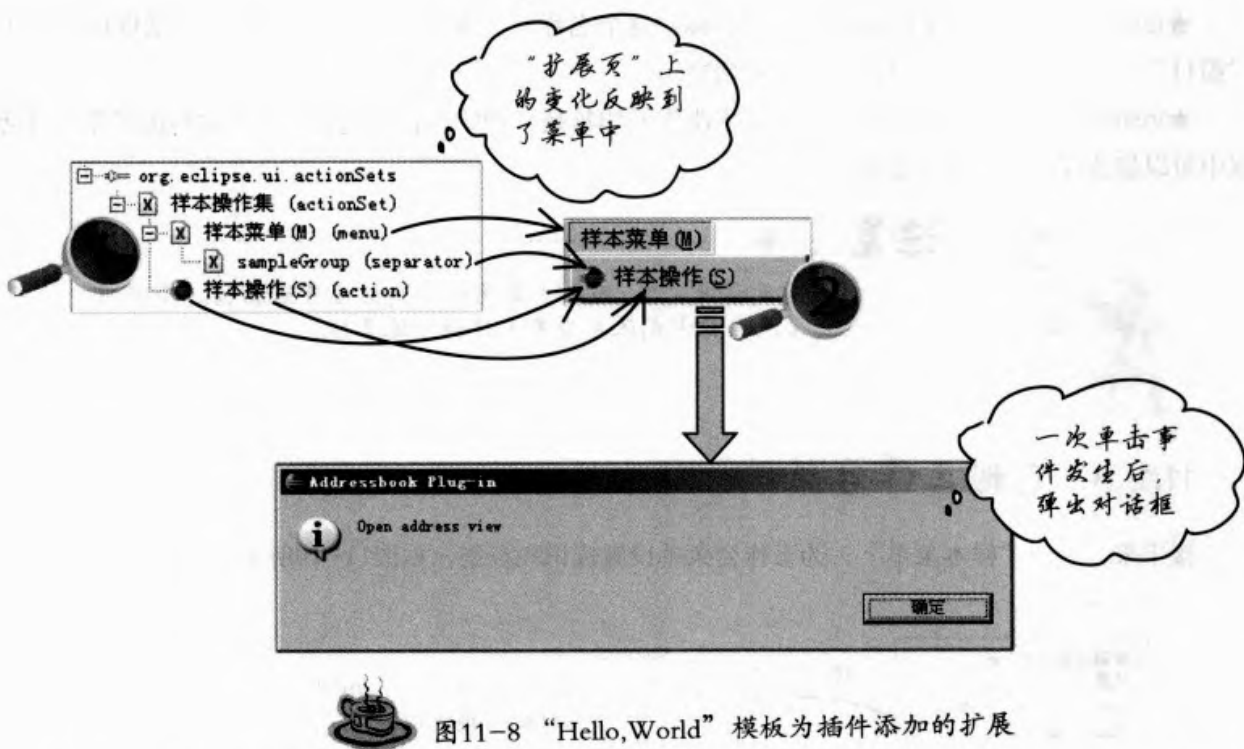


图11-7 弹出的“Addressbook Plug-in”对话框

现在来研究一下模板为开发者做了什么。打开清单编辑器的“扩展”页，看到增加了一个actionSet扩展，这个扩展创建了一个样本操作集，在操作集中又创建了一个样本菜单和一个样本操作，如图11-8所示。



在图11-8中，显示了扩展点的神奇之处。模板在plugin.xml文件中添加了一个“样本操作集”扩展点。仅仅为其增加了一些XML代码（在“扩展页”中以图中处所示的树型结构表示），便在程序中增加了新的顶层菜单和操作，如图中处所示。

模板还为插件开发者添加了一些代码来定制行为，弹出的“Open address View”对话框便是用代码实现的。

11.2.2 定制操作集

修改产生的清单文件，使其匹配地址本插件。首先，在扩展选项卡中，选中“样本操作集”，在“扩展详细信息”选项卡中，设置其属性；或者直接在plugin.xml文件中修改，如图11-9所示。

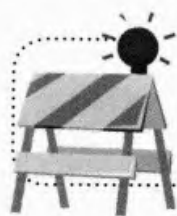


工作集中的属性及其含义如下所示。

★id*——引用该操作集所使用的唯一标识符。

★label*——用户可见的该操作集的名称，这个名称会出现在“定制透视图”对话框中（使用“窗口”→“定制透视图”可打开这个对话框）。

★visible——该属性使用布尔值，用来决定操作集是否初始状态下可见。在“定制透视图”对话框中可以修改操作集的可见状态。



注意：

后面标有“*”的是必要属性，其他为可选属性。本书将一直采用这种方式区分必要属性和可选属性。

11.2.3 定制工作台菜单

接下来，选中“样本菜单”，为工作台菜单设置标识和标签，如图11-10所示。

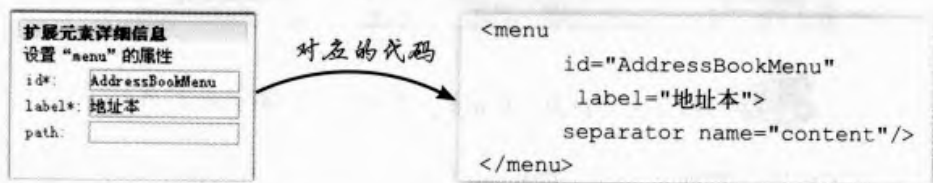


图11-10 设置地址本菜单的属性

样本菜单中的属性及其含义如下所示。

★id*——用于这个菜单的唯一标识符。

★label*——出现在菜单栏中的菜单名称。

★path——表示在菜单栏中何处放置菜单的插入点。

注意：



插入点用来定义菜单出现的位置。由于Eclipse由插件组成，而每个插件又都可以添加操作，这些插件之间可能不会彼此知道。为此，Eclipse仿照标识符的使用方式，使用“插入点” (insertion point) 来指定菜单或操作将在工作台的什么位置出现。

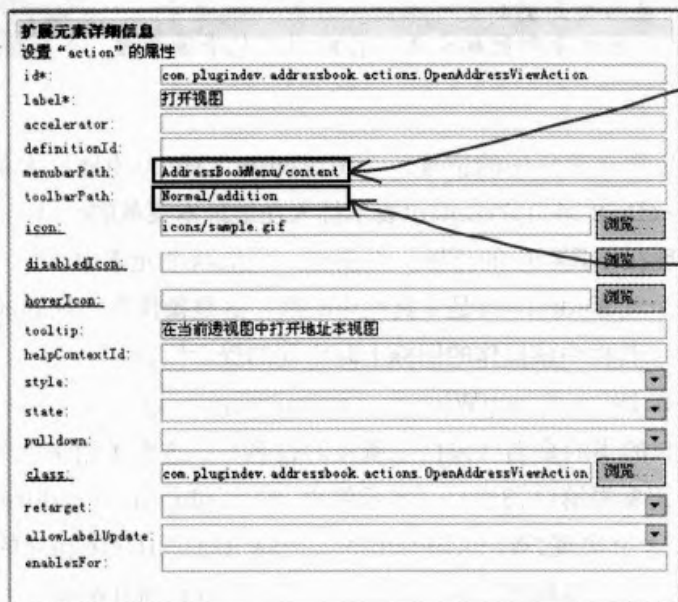
Eclipse系统中定义了标准的插入标识符addition，用来将新创建的操作插入指定的位置，这个标识符作为常量定义在org.eclipse.ui的IWorkbenchActionConstants中，可以使用IWorkbenchActionConstants.MB_ADDITIONS来获得。例如，假如希望将某个菜单放入“帮助”菜单中，就应该将其属性设置为help/additions。



在菜单中，还包含有separator元素和groupMarker元素，用来分离不同的组，它们只有一个必要属性name。创建separator会在组中增加一条水平线，而groupMarker不会添加水平线。

11.2.4 定制操作菜单项

最后，将要定制出现在“地址本”菜单和工作台工具栏中的操作。选中“扩展”选项卡中的“样本操作”，图11-11所示，可在其中更改相应的值。



操作将在地址本菜单的子菜单中显示

操作还将在工具栏中显示，将出现在最后一个工具栏按钮之后



图11-11 设置“action”的属性

这些属性的含义如下所示。

★id*——引用这个操作的唯一标识符。

★label*——用户可见的出现在“地址本”菜单中的文本。

★icon——与该操作相关联的图像。可以使用“浏览”，在弹出对话框中选择icon文件夹中的sample.gif，这是视图模板提供的。如图11-12所示。



图11-12 定义操作相关联的图像

* 注意: *

可以用Turbo Icon Editor或Adobe Photoshop Element创建自己需要的图标。还有一种图标提取工具 EXE 图标提取器, 可以从应用程序中提取图标。Eclipse的所有图标集可以在Eclipse安装目录下的plugins\org.eclipse.ui_3.0.0\icons\full中找到。

向插件添加图标可以选择插件中的icons文件夹, 右击选择“导入”, 在向导中选择选择“常规”→“文件系统”, 然后找到图标文件存储的目录, 按选中, 单击“完成”按钮。或者直接在文件系统中复制图标文件, 粘贴到“包资源管理器”中的icons文件夹即可。

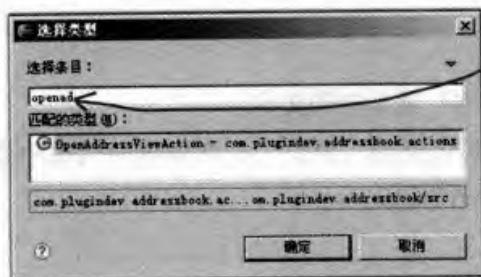


★menubarPath——表示操作将出现在菜单中的位置。AddressBookMenu为地址本菜单的标识, content为separator的名字。AddressBookMenu/content表示插入在地址本菜单的content中。

★toolbarPath——表示操作将出现在工具栏中的位置。在斜杠分隔的两个元素中, 第一个元素Nomal为工作台工具栏的标识符, 第二个元素additions是工具栏中的组, 这是操作将出现的地方。

★tooltip——当鼠标悬浮在工作台工具栏中该操作的图标上时出现的提示信息。

★class——一个实现org.eclipse.ui.IWorkbenchWindowActionDelegate或org.eclipse.ui.IWorkbenchWindowPulldownDelegate接口的类的全名(包括它所在的包名), 这个类用来作为Action的代理(关于Action和ActionDelegate请参看第11.3节)。本示例中 com.plugindev.addressbook.actions.OpenAddressViewAction这个实现IWorkbenchWindowActionDelegate的类是由“Hello, World”模板创建的。如果指定了下拉样式, 那么就需要实现IWorkbenchWindowPulldownDelegate。如果“retarget”的值为true, 则该属性将被忽略。可以通过“浏览”按钮, 在弹出的对话框中输入类名(不需要输入全名)来选择需要的代理类, 如图11-13所示。



在此输入类名后
Eclipse会进行自
动匹配



图11-13 选择匹配的代理类



另外, 还有一些选填项, 它们的含义分别如下所示。

★definitionId——操作的命令标识符, 它允许将按键绑定到操作上。

★disabledIcon——禁用操作时相应的图像。有些操作需要在禁用操作时图像变灰, 这需要设置disabledIcon的值。

★hoverIcon——当鼠标悬浮在操作上但没有单击操作时显示的图像。

★retarget——重定目标操作时需要的属性。当视图或编辑器中需要使用全局工作台菜单或工具

栏中的操作时，将retarget属性设置为true。

★allowLableUpdate——可选，当retarget为true时才起作用。它表示可以重定目标的操作是否允许处理器覆盖它的标签和工具提示。

★Style——定义操作的可见形式的属性。可以在如表11-1所示的值中选择一个。

表11 1 Style属性中规定的值

值	描述
push	普通菜单或工具栏
radio	单选按钮样式的菜单或工具栏项，其中一个在组中，所有单选样式的项一次只有一个项是活动的
toggle	选择的菜单项或切换的工具项。参见state属性
pulldown	子菜单或下拉工具栏菜单。参见class属性

★State——布尔值。对于具有radio或toggle样式的操作，设置其初始状态。

★enablesFor——表示什么时候启动操作。如果此值置空，那么操作总会是活动的，除非通过IAction接口以编程的方式进行覆盖。enablesFor属性支持的格式如表11-2所示。

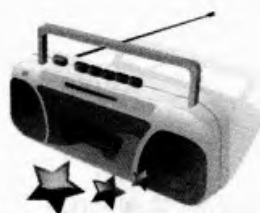
表11 2 enablesFor属性中规定的值

语法	描述
!	选中0个项
?	选中0个或1个项
+	选中1个或多个项
multiple, 2+	选中2个或多个项
n	指定选中项的精确数量
*	选中任何数量的项

在属性中，accelerator和pulldown都已废弃，它们分别由definitionId和style的功能代替。

11.2.5 实现操作代理类

定义完插件扩展点后，打开与之相关联的OpenAddressViewAction类。OpenAddressViewAction实现了“Hello, World”模板已经为开发者创建好了类中的各种方法。如图11-7所示的运行新Eclipse实例所弹出的信息框便是由该类中的run()方法创建。



但这并不符合最初的设想, 最终希望的是打开“地址本视图”而不是弹出一个信息框。为此, 需要在OpenAddressViewAction类中找到run()方法, 更改其中的代码如下所示。

```
public void run(IAction action) {
    //
    if(window == null)
        return;
    IWorkbenchPage page = window.getActivePage();
    if(page == null)
        return;

    //
    try{
        page.showView("com.plugindev.addressbook.views.AddressView");
    }
    catch(PartInitException e)
    {
        MessageDialog.openError(window.getShell(),
            "地址本插件", "打不开地址本视图");
    }
}
```

获得当前页面

打开地址本视图

保存代码后, 重新运行“Address Book”实例, 在新Eclipse工作台中选择工具栏中的打开视图操作, 结果如图11-14所示。同样, 使用“地址本”→“打开视图”操作也可以打开“地址本视图”, 它们调用的是同一个ActionDelegate类。

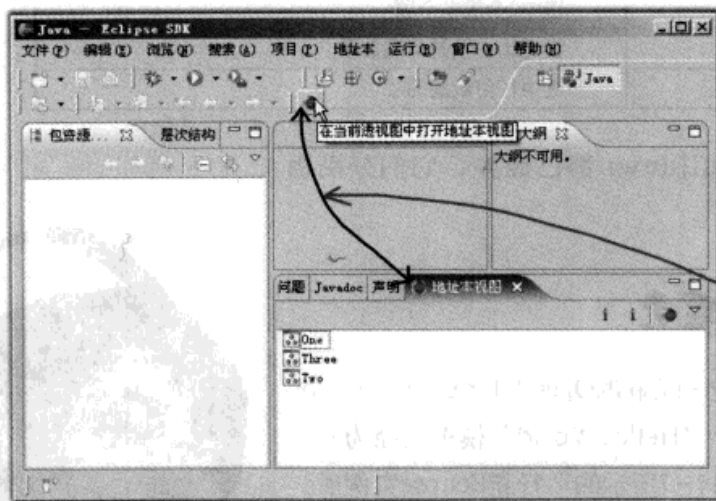


图11-14 使用工具栏操作打开“地址本视图”

11.3 IAction与IActionDelegate接口

第11.1节的例子演示了如何为工作台窗口添加操作, 那么, 这个功能是如何实现的呢? 所创建的

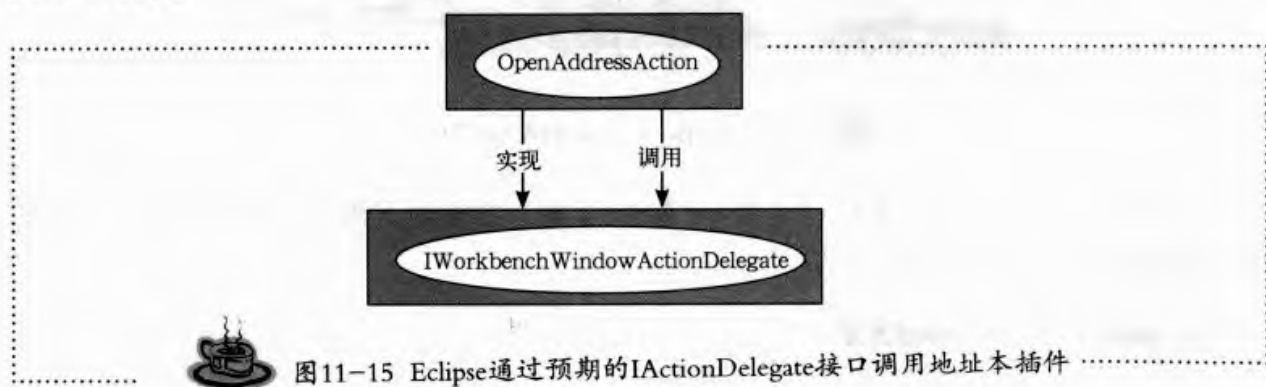
类又是如何起作用？要弄清楚这两个问题，需要了解IAction和IActionDelegate接口。

在Eclipse中，接口的名称一般以“I”开头。Eclipse以这种方式区分接口和与之对应的实现类。IAction与IActionDelegate便是这种命名规则的两个例子。

在第11.2节中已经看到，IActionDelegate接口的子类型IWorkbenchWindowDelegate为工作台窗口的菜单栏和工具栏提供生命周期事件，在后面几节中，将会陆续看到IActionDelegate的其他几个子类型，如下所示。

- ★IEditorActionDelegate——为编辑器中的操作代理提供生命周期事件。
- ★IObjectActionDelegate——为快捷键相关联的操作代理提供生命周期事件。
- ★IViewActionDelegate——为视图中的操作代理提供生命周期事件。

Eclipse的操作由几个部分组成，包括插件清单中操作的XML声明、用来代表这个操作的由Eclipse UI实例化的IAction对象，以及在插件清单中定义的实现了IActionDelegate的类。在地址本插件中定义的类OpenAddressViewAction即为实现IActionDelegate的类的一个例子，它们之间的关系如图11-15所示。



Eclipse工作台根据清单中描述的信息（标题、图标、工具提示等）实例化一个IAction对象（WWinPluginAction，继承了抽象类PluginAction，该抽象类可以将Action对象变为真实动作的代理），这个对象具备操作的所有信息，但它唯一不能做的事便是执行实际操作。图11-16说明了Action和IActionDelegate二者的关系。

执行操作的任务由具体的实现了IActionDelegate的类来定义，在图11-16中，这个类是OpenAddressViewAction类。当用户单击工具栏按钮时，PluginAction对象会加载这个类（依据图中①处的描述信息来获得该类的名称），并且调用其run()方法，使操作真正得到执行。Eclipse通过这种方式成功地将显示与实现相分离，延迟了对类对象的加载。

※ 注意：※

这种调用模式是Self Delegation模式，真正的操作对象以参数的方式接受代理操作对象，从而获得操作的状态。代理操作对象通过这个途径，成功地将任务转移给真正的操作对象。关于Self Delegation模式的更多内容，读者请参阅K.Beck所著的《Smalltalk Best Practice Pattern》一书。



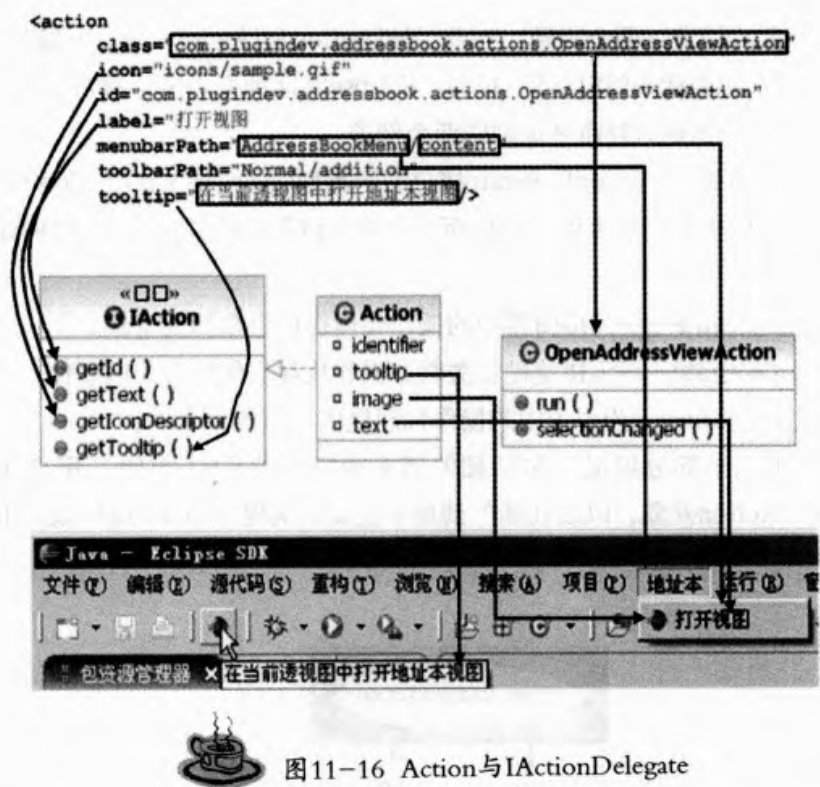


图11-16 Action与IActionDelegate

现在来关注一下OpenAddressViewAction类，它有如下两个主要的方法，这也是IActionDelegate接口提供的两个方法。

1. selectionChanged方法

插件清单中的操作声明提供了操作的初始状态，而操作代表中的selectionChanged()方法提供了一个使用IAction接口动态的调整状态，启动操作的机会。例如，enablesFor属性（参见第11.2.4节）用于指定对象的数量，选择一个要启动的操作，但是通过selectionChanged()方法，可以提供该启用进一步的精细功能。利用这个方法可以询问当前的选择，并且在必要时调用IAction.setEnabled()方法，从而重新启用该操作。

2. run方法

当用户在有效状态下选择某个操作后，该方法将被调用。一般情况下，selectionChanged()方法会在run()方法之前执行，以便判断选择的状态。但是，如果该操作第一次被用户选择，而且提供该操作的插件在此之前未被加载，那么选择这个操作会导致插件被加载，这样，就会直接调用run()方法。

11.4 对象操作

本节和第11.5节主要介绍在Eclipse视图中创建操作的方法。在Eclipse视图中创建的操作可以分为

两种，一种是指向特定对象类型的操作，另一种是在视图的专用菜单中提供的操作。

对于地址本插件，可以设想，当这个插件功能完善之后，应该很容易支持用户管理信息，包括添加、删除、修改、分组等功能。基于对象的操作对此非常理想，它支持用户在只有在视图或编辑器中选择了这个对象的情况下，这些功能才出现在上下文菜单中。Eclipse的“包资源管理器”视图的树状结构中的操作就是基于对象的，如图11-17所示，不同类型的对象的上下文菜单不同。

在本节的后续部分，将通过一个实例来介绍如何用插件的方式来创建对象操作。在当前的“地址本视图”中，上下文菜单操作有Action1和Action2两个，这两个操作都是对所有对象通用的，而且是以编程方式实现的（参见第11.5节）。本节将为名称为“three”的对象添加一个objExampleAction操作。

※ 注意 : ※

Action1和Action2是视图模板为开发者生成的，它们是通过编程的方式实现的，和本节将要实现的objExampleAction的实现方式有所不同。了解如何通过编程方式实现操作，请参见第12章。

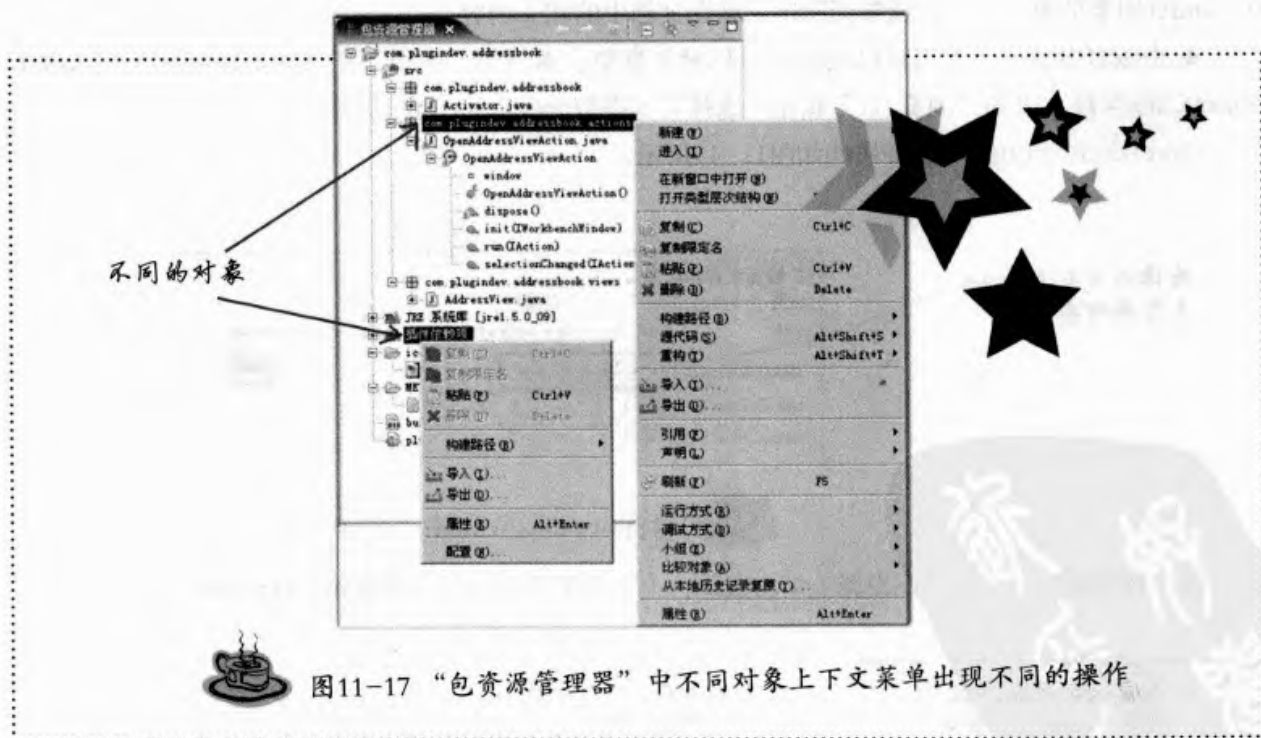


图11-17 “包资源管理器”中不同对象上下文菜单出现不同的操作

11.4.1 添加对象操作

当前的“地址本视图”包含三个表格元素（TableItem），它们分别表示三个String对象，One, Two, Three。可以使用插件清单编辑器的“扩展”页面创建新的对象添加行为。单击“添加”

按钮, 添加org.eclipse.ui.popupMenus扩展, 然后选中该扩展, 在上下文菜单中选择“新建”→“objectContribution”, 如图11-18所示。

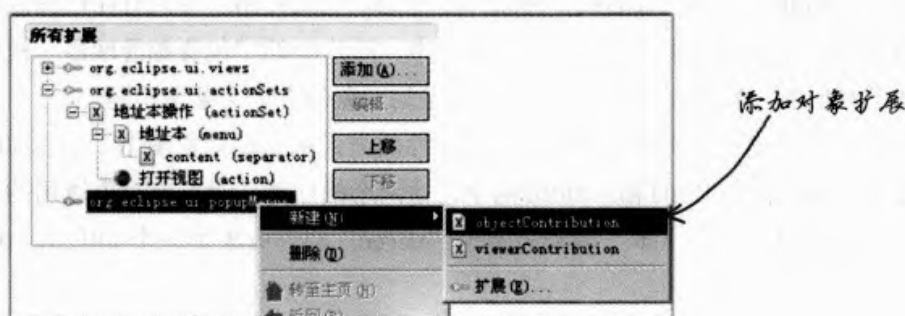


图11-18 添加一个objectContribution

objectContribution元素有4个属性, 它们分别如下所示。

★id——添加此操作的唯一标识符。

★nameFilter——通配符过滤器, 该过滤器指定可以接受目标的名称。例如, 输入“*.java”将只包括那些名称以“.java”结尾的文件。本例中, 设置nameFilter的值为“Three”。

★adaptable——指出适配IResource的对象是可接受的目标, 默认该属性为“False”。关于IResource对象的更多信息, 请参阅Eclipse帮助文档中的相关内容。

★objectClass——指出可以接受的目标对象类型。本例中, 使用“扩展元素详细信息”选项卡objectClass字段右边的“浏览...”按钮, 选择基本类型java.lang.String类。

objectContribution的各种属性如图11-19所示。

将操作贡献到String
类型的对象上

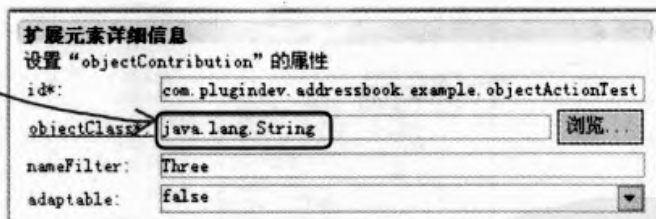


图11-19 objectContribution的属性

接下来同第11.2节类似, 设置上下文菜单, 上下文菜单中的各种属性如图11-20所示。



图11-20 String对象上下文菜单的属性信息

在创建的上下文菜单中，添加一个separator，设定其名字为example。

下一步，在新的objectContribution中添加一个操作ObjExampleAction。新操作的属性值都同第11.2节中的类似，其属性信息如图11-21所示。这里为enablesFor属性值设定为“1”，表示只有选中一个对象时，该操作才启用（参见第11.2.4小节）。



图11-21 “对象示例操作”的属性值

11.4.2 操作的可见性

objectContribution有一个子元素visibility，该元素提供了替代nameFilter和ObjectClass的方法，并且更为强大。例如，上面描述的指定对象添加的过滤的一种替代方式可以如下所示。

```
<objectContribution>
.....
  <visibility>
    <objectClass
      name="java.lang.String"/>
  </visibility>
.....
</objectContribution>
```



<visibility>元素的属性如下所示。

★objectClass——将选中对象的类同name的值相对比，比较结果为真，则显示该操作。

★objectState——将选中对象的状态同上面显示的指定状态进行对比，它具有名字和值两个属性。

★pluginState——该元素有id和value两个属性。id是指相关联的插件的标识符，value可取的值有两个，一个是installed，另外一个为activated。installed是指当安装了id所标示的插件后，操作才

可见，activated是指仅当id所标识的插件安装并且已经以某种方式激活时，操作才可见。默认选择为installed。

★systemProperty——该元素用来对系统的一些属性做出适配。例如，如果需要在高级模式下使用该操作，表达式应该如下所示。

```
<systemProperty name="ADVANCED_MODE" value="true"/>
```

11.4.3 操作的过滤

filter元素是visibility元素中的objectState的一种替代形式，但它不用包含在visibility元素内。如要使对优先级为1的对象使用该操作，则应该使用如下表达式。

```
<filter name="priority" value="1"/>
```

filter和objectState均使用IActionFilter接口，决定着选择的对象是否匹配条件。每个选中的对象必须实现或适配IActionFilter接口，并且实现其中的testAttribute()方法，以便测试相应的“名/值”对。

另外，还有其他不同的形式对操作进行过滤，如使用selection元素。selection元素是基于操作的名称和类型，启用单个操作的技术。它的方式类似于nameFilter和objectClass属性，决定在对象添加中的操作是否可见。例如，对象添加使用selection元素的另一种形式如下面代码所示。

```
<objectContribution
  objectClass="java.lang.Object"
  id="com.plugindev.addressbook.example.objectActionTest">
  <action
    label="对象示例操作"
    tooltip="举例说明如何添加对象操作"
    class="com.plugindev.addressbook.example.ObjExampleAction"
    menubarPath="additions"
    enablesFor="1"
    id="com.plugindev.addressbook.objExampleAction">
    <selection
      class="java.lang.String"
      name="three"/>
    </action>
  </objectContribution>
```



11.4.4 实现IObjectActionDelegate接口

前面将操作指向了一个类ObjExampleAction，它必须实现IObjectActionDelegate接口。IObjectActionDelegate与第11.2节和第11.3节介绍的IWorkbenchActionDelegate非常相似。

首先，需要创建一个名为com.plugindev.addressbook.example的包，在这个包里将会包含插件中作为示例的所有代码。然后再创建一个实现IObjectActionDelegate的类ObjExampleAction，选择“来自超类的构造函数”和“继承的抽象方法”，如图11-22所示。

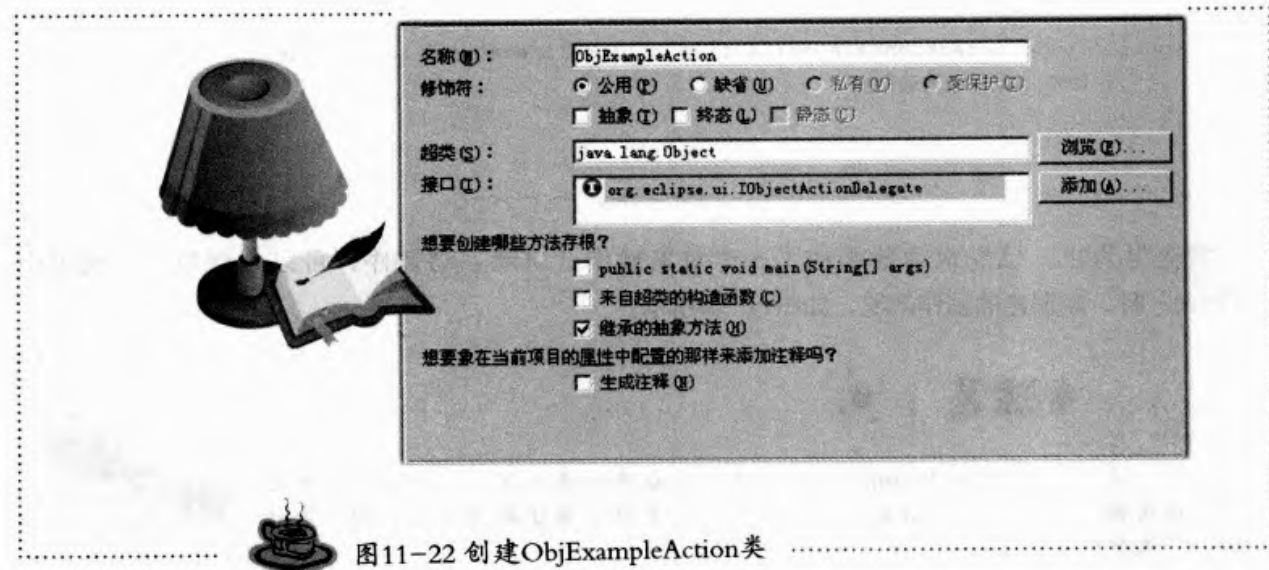


图11-22 创建ObjExampleAction类

在ObjExampleAction类中，生成了setActivePart(),selectionChanged(),run()三个方法，如图11-23所示。其中，selectionChanged()方法和run()方法的作用和OpenAddressView Action中的这两个方法相同。当每次操作在上下文菜单中出现时setActivePart()被调用，用来为代理设置活动部分。

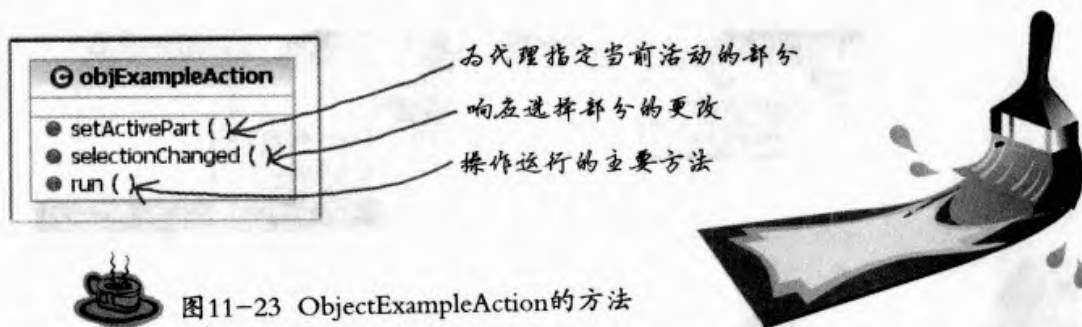


图11-23 ObjectExampleAction的方法

为了说明操作已得到执行，现在在run()方法中创建一个弹出信息框。下面列出了ObjExampleAction的代码。

```
package com.plugindev.addressbook.example;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IWorkbenchPart;
public class ObjExampleAction implements IObjectActionDelegate {
    private IWorkbenchPart targetPart;
    public void setActivePart(IAction action, IWorkbenchPart targetPart) {
        //
        this.targetPart = targetPart;
    }
    public void run(IAction action) {
        MessageDialog.openInformation(targetPart.getSite().getShell(),
            "消息", "对象示例操作被执行");
    }
}
```

为代理设置活动部分，
每次操作在上下文菜单
中出现时被调用

显示一条操作已
执行的消息


```

    }
    public void selectionChanged(IAction action, ISelection selection) {
        // TODO 自动生成方法存根
    }
}

```

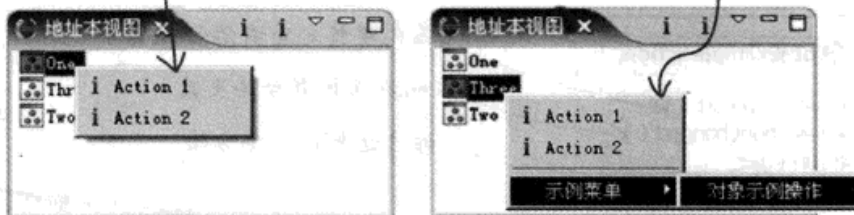
到这里为止，已经成功地添加了一个对象操作。重新启动插件，可以看到当选中视图中“Three”时，新创建的操作出现，如图11-24所示。

※ 注意 : ※

在没有创建`ObjExampleAction`便运行程序的情况下，编译器不会提示任何错误信息，当单击新创建的操作时，会弹出系统提供的对话框，提示“该操作目前不可用”。



不同的对象出现了不同的操作
不同的对象出现了不同的操作



11-24 当选中“Three”时“对象示例操作”出现

选择该操作，则`run()`方法被调用，弹出信息对话框，如图11-25所示。

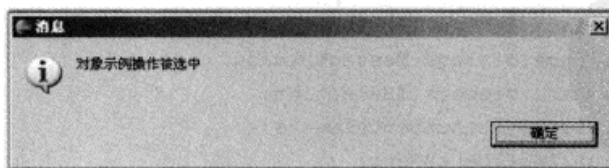


图11-25 `ObjExampleAction`中的`run()`方法被调用

11.5 视图操作

由图11-1可知，视图操作包括视图上下文菜单，视图工具栏按钮、出现在视图工具栏的下拉菜

单。这些操作都可以通过同前面三节中介绍的扩展点机制来实现。而且，视图中所有类型的操作均可通过编程的方式实现（参见第12章）。

11.5.1 添加视图的上下文菜单

同objectContribution不同的是，viewerContribution不是针对视图中的对象决定应出现的上下文菜单，而是根据查看器（viewer，将在第12章介绍）的类型而选择，这也就意味着，viewerContribution规定的是针对该视图中所有对象的通用操作。

viewerContribution与第11.4节介绍的objectContribution大同小异，它也是popupMenus的一个子元素，因而，作为popupMenu的一部分，在第11.4节定义的objectContribution之后添加。

由于viewerContribution的大部分元素都跟objectContribution相似，添加扩展和元素、编辑属性的方法也同前面三节中介绍的类似，在此不再一一赘述，本章后续部分将直接对plugin.xml进行文本编辑。打开plugin.xml页面，在其中添加如下所示的XML代码。

```
<viewerContribution
    id="com.plugindev.addressbook.example.viewerActionTest"
    ①targetID="com.plugindev.addressbook.views.AddressView">
    <menu
        id="com.plugindev.viewToolbarMenu"
        label="示例工具栏菜单"
        path="additions">
        <separator name="example"/>
    </menu>
    <action
        ③class="com.plugindev.addressbook.example.ViewerExampleAction"
        ②enablesFor="+"
        id="com.plugindev.addressbook.viewerExampleAction"
        label="查看器示例操作"
        ④menubarPath="com.plugindev.viewToolbarMenu/example"
        tooltip="举例说明如何添加查看器操作"/>
    </viewerContribution>
```

在上面的代码中，①处表示的是目标视图的标识符，表示该viewerContribution将为哪个视图提供操作。这里使用的是地址本视图的标识（可在org.eclipse.ui.views扩展中的“地址本视图”中查到）。

在②处的enablesFor的值设定为“+”，这表示此操作对一个或一个以上的对象有效。当按住ctrl键选择多个对象时，该操作仍然出现（有关enablesFor请参看表11-2）。

③处class必须实现org.eclipse.ui.IViewerActionDelegate接口，操作才能得到执行。

在④处的menubarPath里设置的值决定了操作在上下文菜单的位置。它将会在创建的viewToolbarMenu内example分隔符之后出现。

11.5.2 添加视图的工具栏

添加视图的工具栏要使用一个新的扩展点org.eclipse.ui.viewActions。这个扩展点包含viewContribution（注意区别viewerContribution和viewContribution，它们是不同的扩展点的不同元素，在不同的位置添加操作）。viewContribution同前面的objectContribution和viewerContribution类似，它用到的属性在objectContribution和viewerContribution中均已介绍过。

在plugin.xml中添加如下代码。

```
<extension
    point="org.eclipse.ui.viewActions">
    <viewContribution
        id="com.plugindev.addressbook.example.viewActionTest"
        targetID="com.plugindev.addressbook.views.AddressView">
        <action
            ①class="com.plugindev.addressbook.example.ViewExampleAction"
            icon="icons/sample.gif"
            id="com.plugindev.addressbook.ViewExampleAction"
            label="视图操作"
            ②style="push"
            toolbarPath="additions"
            tooltip="举例说明如何添加视图操作"/>
        </viewContribution>
    </extension>
```

①处的class属性仍然需要是实现IViewActionDelegate的类，这点同上下文菜单操作的要求相同。

②处的style的值为“push”，表示添加的是一个普通的菜单（参见表11-1对style各个值的描述）。

11.5.3 添加视图的下拉子菜单

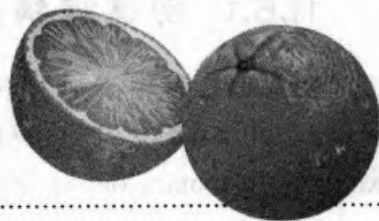
视图的下拉子菜单一般进行过滤、排序等操作。添加视图下拉子菜单的操作很简单，只需要定义一个菜单，将其path设为additions，然后在第11.5.2节中创建的视图操作中加入menubarPath便可以了。在下面的内容中，添加的代码用加粗字体表示。

```
<extension
    point="org.eclipse.ui.viewActions">
    <viewContribution
        id="com.plugindev.addressbook.example.viewActionTest"
        targetID="com.plugindev.addressbook.views.AddressView">
        <action
            class="com.plugindev.addressbook.example.ViewExampleAction"
            icon="icons/sample.gif"
```

```

id="com.plugindev.addressbook.viewExampleAction"
label="视图操作"
menubarPath="com.plugindev.addressViewPullDownMenu/example"
style="push"
toolbarPath="additions"
tooltip="举例说明如何添加视图操作"/>
<menu
id="com.plugindev.addressViewPullDownMenu"
label="下拉菜单示例"
path="additions">
<separator name="example"/>
</viewContribution>
</extension>

```



11.5.4 实现 IViewActionDelegate 接口

实现 IViewActionDelegate 的接口和实现 IObjectActionDelegate 接口类似，只是由于实现的是 IViewActionDelegate 接口，因而没有 setActivePart() 方法，增加了 init() 方法。要想弹出对话框，在 init() 方法中添加如下代码，并更改 run() 方法中弹出的信息对话框信息，其他方面均不变即可。如下所示的是 ViewerExampleAction 的部分代码，在 ViewExampleAction 中的代码与此类似。

```

public class ViewerExampleAction implements IViewActionDelegate {
    private IWorkbenchPart targetPart;
    public void init(IViewPart view) {
        this.targetPart = view;
    }
    .....
}

```

本例中的 ViewerExampleAction 类和 ViewExampleAction 类均被创建在 Java 包 com.plugindev.addressbook.example 中。

添加完针对视图的各种操作后，重启程序，可以看到各种操作都已在视图中出现，如图 11-26 所示。

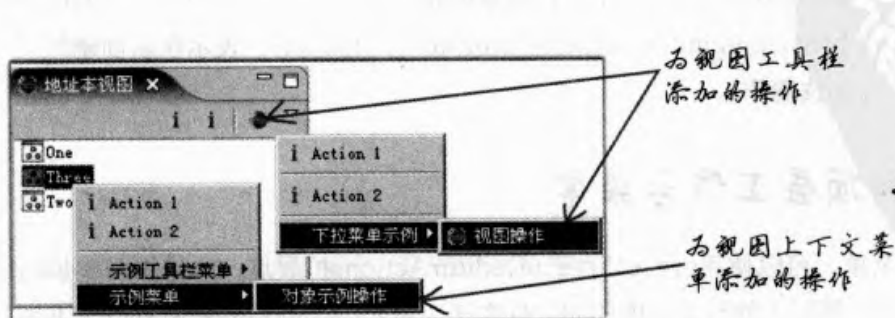


图 11-26 为地址本视图添加完各种操作后的效果图

11.6 编辑器操作

编辑器也支持用扩展的方式添加操作，这跟视图的操作类似。关于编辑器操作的更多细节，请参阅第13章的“为编辑器添加操作”一节。本节介绍使用扩展方式为编辑器提供操作的方法。

11.6.1 创建编辑器上下文操作

要在编辑器中添加上下文菜单项，仍然需要使用org.eclipse.ui.popupMenus的viewerContribution元素，只要将其中的TargetID写为编辑器上下文菜单的标识符即可，在plugin.xml文件中的popumMenu扩展里添加如下代码。

```
<viewerContribution
    id="com.plugindev.addressbook.example.editorActionTest"
    ①targetID="#TextEditorContext">
    ②<menu
        id="com.plugindev.defaultEditorMenu"
        label="示例编辑器菜单"
        path="additions">
        <separator name="example"/>
    </menu>
    <action
        class="com.plugindev.addressbook.example.EditorExampleAction"
        icon="icons/sample.gif"
        id="com.plugindev.addressbook.editorExampleAction"
        label="编辑器操作"
        menubarPath="com.plugindev.defaultEditorMenu/example"
        tooltip="举例说明如何添加编辑器操作"/>
</viewerContribution>
```



①处所示的targetID的值为“#TextEditorContext”，这是Eclipse默认文件编辑器的上下文菜单标识符。

②处创建了一个名为defaultEditor的菜单，该菜单的路径标定为“additions”，创建的编辑器操作使用的menubarPath 为com.plugindev.defaultEditorMenu/example，表示在新创建的菜单中的分隔符“example”之后添加该操作。

11.6.2 添加顶层工作台菜单

添加顶层工作台菜单，可以使用org.eclipse.ui.editorActions扩展点。为编辑器添加的顶层工作台菜单与第11.2节介绍的顶层工作台菜单的不同之处在于，第11.2节介绍的是所有情况下均可见的工作台菜单，而为编辑器添加的顶层工作台菜单只有当使用与之关联的特定的编辑器时才会出现。

单击清单编辑器“扩展”页面的“添加”按钮，添加一个org.eclipse.ui.editorActions扩展，然后为这个扩展添加editorContribution元素。在该元素的“扩展元素详细信息”中，输入如图11-27所

示的id和targetID。

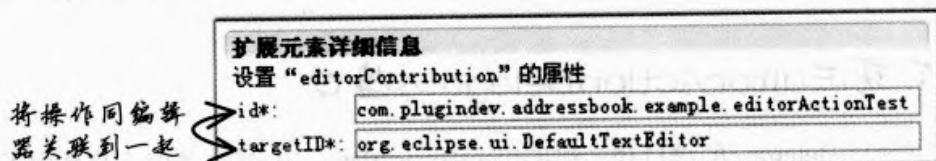


图11-27 editorContribution的属性

targetID是一个必选属性，它用来与特定的编辑器关联，以便此处描述的编辑器打开时，出现editorContribution元素中描述的菜单和操作。

再在editorContribution中创建一个菜单，其属性设置如下面代码所示。

```
<menu
    id="com.plugindev.defaultEditorMenu"
    label="顶层编辑器菜单"
    path="additions">
    <separator name="example"/>
</menu>
```

11.6.3 定义编辑器顶层操作

定义编辑器顶层操作的方法，所用到的属性同定义其他位置的操作没有什么不同。只是需要实现的接口必须为IEditorActionDelegate接口。代码如下所示。

```
<action
    class="com.plugindev.addressbook.example.TopEditorAction"
    id="com.plugindev.addressbook.topEditorAction"
    label="顶层编辑操作"
    ①menubarPath="com.plugindev.defaultEditorMenu/example"
    style="push"/>
```

同样，①处的menubarPath的值表示将操作放在菜单中。

11.6.4 添加编辑器工具栏操作

类似于工作台菜单操作，可以为编辑器定制工作台工具栏按钮。只要修改在第11.6.3节定义的编辑器操作，添加一些属性就可以了，代码如下所示。

```
<action
    class="com.plugindev.addressbook.example.TopEditorAction"
    icon="icons/sample.gif"
    id="com.plugindev.addressbook.TopEditorAction"
    label="顶层编辑操作"
    menubarPath="com.plugindev.defaultEditorMenu/example"
    style="push"
    toolbarPath="Normal/additions"
    tooltip="编辑器操作示例"/>
```


清单中添加的属性有icon,toolbarPath,tooltip。toolbarPath制定了在工具栏中出现的位置。Normal是工具栏的标识符。Normal/additions表示将工具栏按钮添加在所有已定义的按钮之后。

11.6.5 实现IEditorActionDelegate接口

实现IEditorActionDelegate的接口和实现IObjectActionDelegate接口类似，只是由于实现的是IEditorActionDelegate接口，因而没有setActivePart()方法，而代替以setActiveEditor()方法。本例中操作执行将会弹出对话框，这需要在setActivePart()方法中添加如下代码，并更改run()方法中弹出的信息对话框信息，其他方面均不变即可。如下所示的是TopEditorAction的部分代码，在EditorExampleAction中的代码与此类似。

```
public class ViewerExampleAction implements IViewActionDelegate {
    private IWorkbenchPart targetPart;
    public void init(IViewPart view) {
        this.targetPart = view;
    }
    .....
}
```

本例中的TopEditorAction类和EditorExampleAction类均被创建在Java包com.plugindev.addressbook.example中。

添加完针对视图的各种操作后，启动应用程序，选择“新建”→“无标题的文件”，程序将会用DefaultTextEditor新建并打开一个文件。这时，可以看到各种操作都已在程序中出现，如图11-28所示。

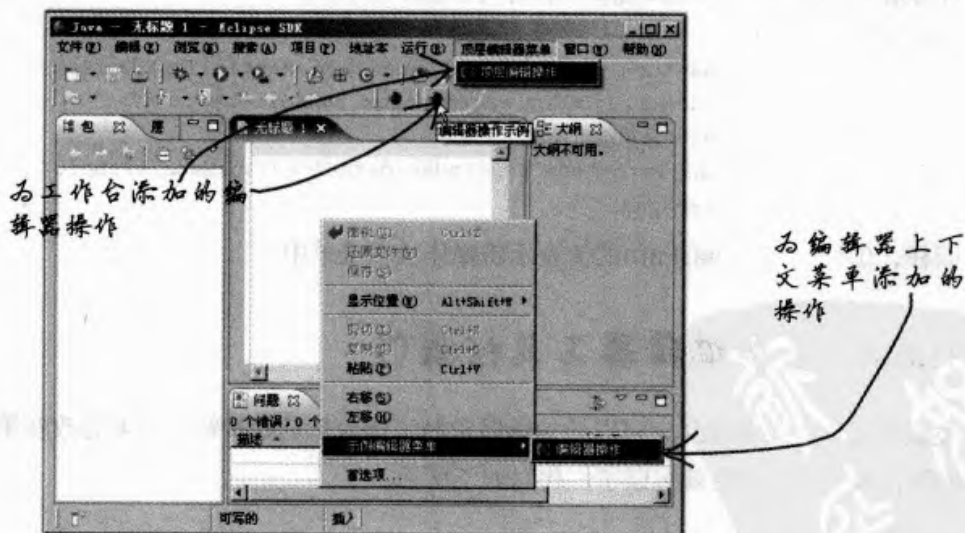


图11-28 为编辑器添加完各种操作后的效果图



11.7 快捷键映射

Eclipse的工作台操作和编辑器操作都支持键盘操作，将操作同键盘的快捷键对应起来。快捷键映射也是既可以用编程方式实现，也可以用在代码文件中添加扩展的方式实现。本节介绍用扩展方式添加快捷键映射的方法。

11.7.1 键绑定的策略

Eclipse的麻烦之处在于，它是由许多不同的开发者贡献的插件集成在一起的。这些开发者在贡献插件之前并没有很好地协商，而且由于Eclipse是开放源代码的，全世界不同角落的开发者都会为Eclipse平台的发展出谋划策，这使得Eclipse平台的更新速度很快，在未来的工作台中可能会添加更多新功能。因而，如果要支持快捷键映射，就必须考虑到以下三个问题。

★不同的插件可能会为不相关的操作定义相同的快捷键。

★插件可能会为相同的操作定义不同的快捷键。

★插件可能会定义与工作台冲突的快捷键。

因而，Eclipse不能直接将操作与快捷键相关联，即不能使操作与快捷键成为一一对应的关系，这样将没有足够的措施来应对上面的三个问题。一个完善的开发工具应该为所有可能出现的冲突或错误提供保障措施，而不能将这些工作都推给开发者。

为了解决这些问题，Eclipse引入了命令（Command）的概念，使用命令来定义语义操作。

11.7.2 创建命令

命令就是操作及其相关联类别的声明。可以使用命令与键绑定、操作和处理程序相关联，但是它并不定义操作的实现，关于操作实现的具体细节由操作本身来完成，正如在本章前面几节中介绍的那样。这样做的好处是将命令从操作实现中分离出来，以便允许多个插件定义实现相同语义命令的操作。

现在介绍一个具体的例子来说明如何创建命令。在创建命令之前，先创建一个类别，以便将命令分组。在插件清单的“扩展”页面中，添加一个“org.eclipse.ui.commands”扩展点。选中该扩展点，右键单击，新建一个类别。设置其属性如图11-29所示。



图11-29 设置category的属性

在命令中使用的`categoryId`为创建的命令类别`id`

扩展元素详细信息
设置“command”的属性

id*:	com.plugindev.addressbook.commands
name*:	打开地址本
category:	
description:	打开地址本视图
categoryId:	com.plugindev.addressbook.commands
defaultHandler:	<input type="text"/> 浏览...
returnTypeId:	
helpContextId:	



图11-30 command元素的属性信息

使用到属性的含义如下所示。

★**id***——命令的唯一标识符。

★**name***——命令的可读性名称。

★**category**——命令的标识符，在3.0以后的版本中，此属性已弃用。用`categoryId`代替。

★**description**——命令的相关描述。

★**categoryId**——在3.0以后的版本中，此属性描述命令的唯一标识符。

如果需要了解更多的属性，在“扩展”页选中`org.eclipse.ui.commands`扩展点后，打开右边的“打开扩展点描述”链接，这里有对该扩展点的所有元素和属性的定义及简单的示例清单。

11.7.3 键绑定

接下来，添加`org.eclipse.ui.bindings`扩展点，在该扩展点中新建`key`元素，为`key`元素设置属性如图11-31所示。

当用户按下`CTRL+SHIFT+V`这三个键后，关联的命令将会执行

扩展元素详细信息
设置“key”的属性

sequence*:	CTRL+SHIFT+V
schemeId*:	org.eclipse.ui.defaultAcceleratorConfiguration
contextId:	org.eclipse.ui.contexts.window
commandId:	com.plugindev.addressbook.commands.openAddressView
platform:	
locale:	



图11-31 设置“key”的属性

使用到的属性含义如下所示。

★**sequence***——指定绑定的键序列。键序列通常由管理键（ALT，COMMAND，CTRL和

SHIFT) 和普通的键组成, 普通的键一般由其指代的ASCII字符的大写形式来表示。不同的键之间以“+”分隔。

除管理键和普通键之外, 还有一些特殊键, 这些特殊键的表示掩码举例如表11-3所示。

表11-3 特殊键的表示掩码

语法	描述
CAPS_LOCK	大写锁定键
ESC	退出键
F1	功能键
PAGE_UP	上翻页
ARROW_DOWN	向下箭头
NUMPAD_1	数字键1

★**schemeId**——激活该键序列使用的配置。在一般情况下, 将键绑定添加到默认的Eclipse配置 `org.eclipse.ui.defaultAcceleratorConfiguration`。

★**contextId**——激活该键序列使用的下文环境的标识符。默认使用的上下文环境标识符为 `org.eclipse.ui.context.windows`。

★**commandId**——该键被激活后, 所触发的命令。

11.7.4 命令与操作关联

完成键设定之后, 还差最后一步, 那就是将命令与操作相关联。在“扩展”页中找到 `org.eclipse.ui.actionSets` 扩展点, 在“地址本操作”中选择“打开视图”, 在右侧的“扩展元素详细信息”中找到“definitionId”属性, 输入创建的命令的标识符 `com.plugindev.addressbook.commands.openAddressView`, 让操作现在引用新的命令和相关联的键绑定。

启动应用程序后, 在顶层“地址本”菜单的子菜单“打开视图”中, 出现了“Ctrl+Shift+V”的标识, 如图11-32所示。



图11-32 添加了键绑定的“打开视图”菜单

使用键盘组合Ctrl+Shift+v三个键同时按下, “地址本视图”将会打开。

当将操作同命令绑定后, 就可以支持操作的定制了。打开“窗口”→“定制透视图”, 选择“命令”选项卡。在“可用的命令组”里可以找到在本章中创建的“地址本操作”, 右边的“菜单栏详细信息”和“工具栏详细信息”中会显示该操作可用的菜单和工具栏操作列表, 如图11-33所示。

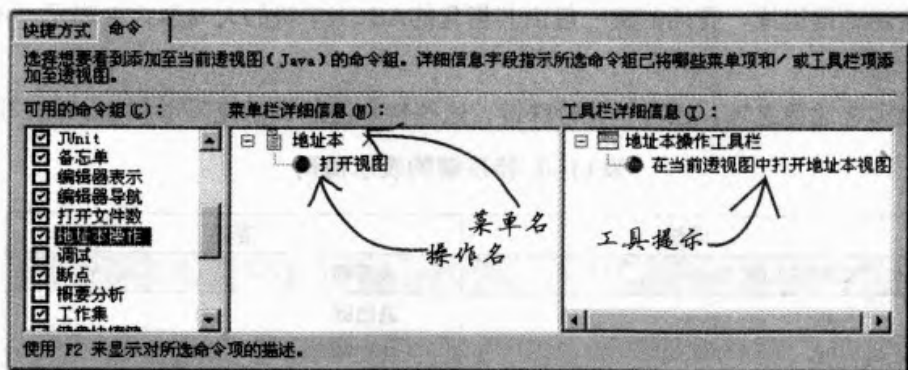


图 11-33 地址本操作出现在“定制透视图”对话框中

命令组的选取标记表示该菜单和工具栏操作在工作台中可视。通过对命令组打勾或取消打勾可以定制在当前透视图中的可见的操作。

11.8 本章小结

本章讨论了如何使用扩展点机制创建各种操作，以及如何将操作映射到键盘上，并简要介绍了命令的概念。尽管本章所列举的操作都非常简单，仅仅是示范性的，许多操作并没有提供真正有实际意义的功能。但是通过这些例子学到的利用扩展点实现操作的内部机制，可以帮助开发者使用选择和更多功能对话框（参见后面的章节）来使这些操作执行更有意义的任务。

操作是用户同插件交互的重要手段，因而，理解并掌握Eclipse中操作的实现机制及操作可以实现的功能是非常重要的。操作可以实现如下所示的功能。

1. 为工作台窗口添加新的行为。
2. 为对象类添加新的行为。
3. 为已存在的视图和编辑器添加新的行为。

我们成功了！哈哈！



第12章 视图 (Views)

拉开崭新的学习帷幕

上一章介绍了如何使用扩展的方式为Eclipse插件添加操作，除工作台操作可以处理全局对象外，每个操作都应该基于一定的对象，如视图操作应该基于视图来进行。本章将在第10章使用模板创建的视图的基础上，通过为地址本插件增加一个新的视图来介绍视图的各方面知识，包括创建视图、修改视图以响应编辑器或其他视图的更改，以及关联视图到其他视图和编辑器。

本章内容包括：

- ★介绍Eclipse视图的体系结构和Eclipse中的各种视图。
- ★创建一个视图，介绍同视图直接相关的视图类。
- ★为视图提供模型底层数据模型，并实现排序、过滤等功能。
- ★使用编程的方式为视图添加操作。
- ★介绍视图间通信的方式，实现属性视图。
- ★为视图添加状态支持，保证用户使用的一致性和连贯性。

进入第12章

12.1 Eclipse视图体系结构概览

在以Eclipse为核心的程序中，视图和编辑器组成了基本界面构件块（building block）。Eclipse的可扩展性表现在不仅允许开发者向现有的视图和编辑器添加功能，还允许添加自己的视图和编辑器。许多插件都通过添加新的Eclipse视图或增强现有的Eclipse视图，向用户提供信息。

在第9章简单介绍了工作台层次结构（参见第9.1.3节“工作台层次结构”），其中提到视图包含在工作台页面中（实现org.eclipse.ui.IWorkbenchPage的类），而工作台页面包含在工作台中；而且，所有视图都必须实现org.eclipse.ui.IViewPart接口，通常作为ViewPart的子类。可以通过如图12-1所示的类图来了解ViewPart的继承和调用关系。

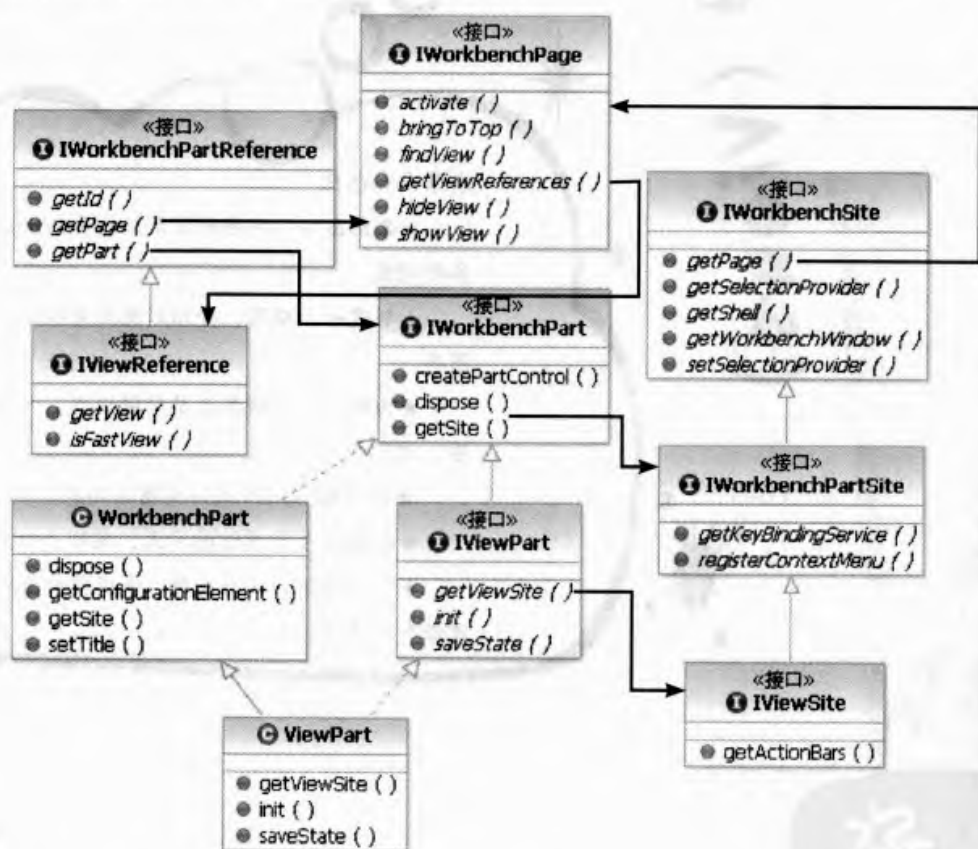


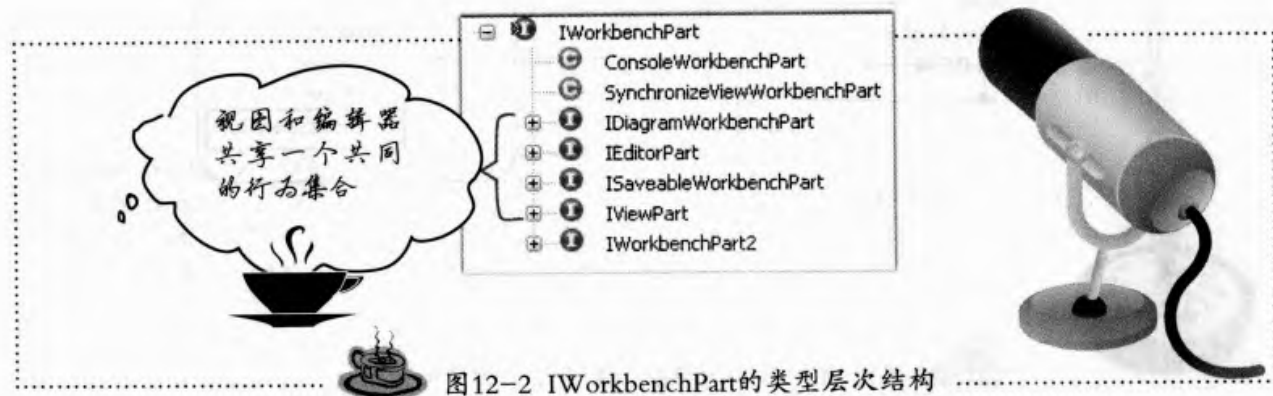
图12-1 ViewPart类图以及调用关系

从图12-1中可以看到，视图包含在视图站点（org.eclipse.ui.IViewSite）中，并通过视图站点包含在工作台页面中。视图通过方法getViewSite()来获得视图站点，而视图站点通过实现继承自工作台站点（org.eclipse.ui.IWorkbenchSite）的方法getPage()来获得工作台页面。

在工作台页面中保存了视图的参考信息（org.eclipse.ui.IViewReference）的实例，通过IViewReference所提供的方法getView()来间接获取视图。这样避免了工作台页面直接保存视图本身，从而使引用视图同加载定义视图的插件相分离，可以避免或延迟相关插件的加载。

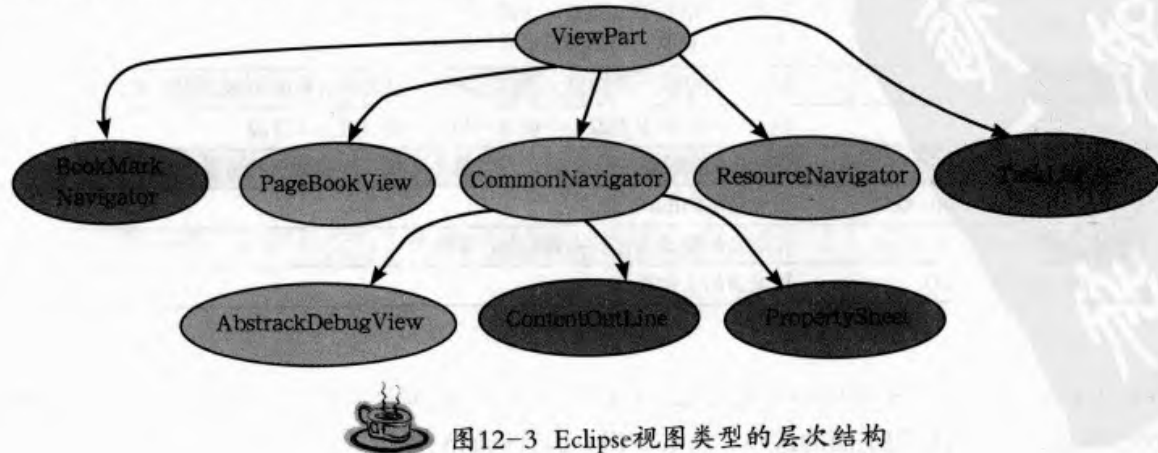
除了IViewPart以外，IEditorPart也继承自IWorkbenchPart，这说明视图和编辑器共享一个共同的行为集合，两者之间有着比较多的相似之处，如图12-2所示。但二者之间也有着很重要的区别，本书在首次使用视图和编辑器时提出，经过后面各章不断完善地址本插件，对这些区别的印象会更为深刻。视图与编辑器的区别如下所示。

- 1.如果有编辑器，编辑器将会出现在Eclipse的中心区域，而视图围绕着编辑器显示。
- 2.视图同资源可以相关，也可以无关，而编辑器一般则是针对资源的，往往涉及对资源（如文件）的操作。
- 3.视图不像编辑器，没有自己的输入源，它一般显示当前正在使用的编辑器或工作台的状态信息。
- 4.视图同工作空间和底层资源联系在一起，视图中的所有操作都会立刻响应工作空间和底层的状态，而编辑器直接操作的则是内存中的副本，在没有保存时，不会写入磁盘，也即不会改变底层资源的状态。



12.2 Eclipse工作环境中的视图

Eclipse工作环境中包含了很多预定义视图，这些视图为开发者提供了多个角度去观察资源或编辑器的状态，为应用程序的开发带来了很大的便利。如图12-3所示是Eclipse中视图类型的层次关系。



在图12-3中，深色的视图表示的是不能被再次实例化也不能拥有子类的视图类型，稍浅一些的“CommonNavigator”可以被实例化，但是不建议继承该类，浅色的表示既可以被实例化也可以被继承的视图类。开发者可以在ViewPart或其可继承的子类的基础上定义自己的视图类。

12.2.1 资源导航

ResourceNavigator类实现了导航器 (Navigator)，提供了查看Eclipse工作空间资源的功能，如图12-4所示。开发者可以通过实现IResourceNavigator接口来应用自己的导航器设置。

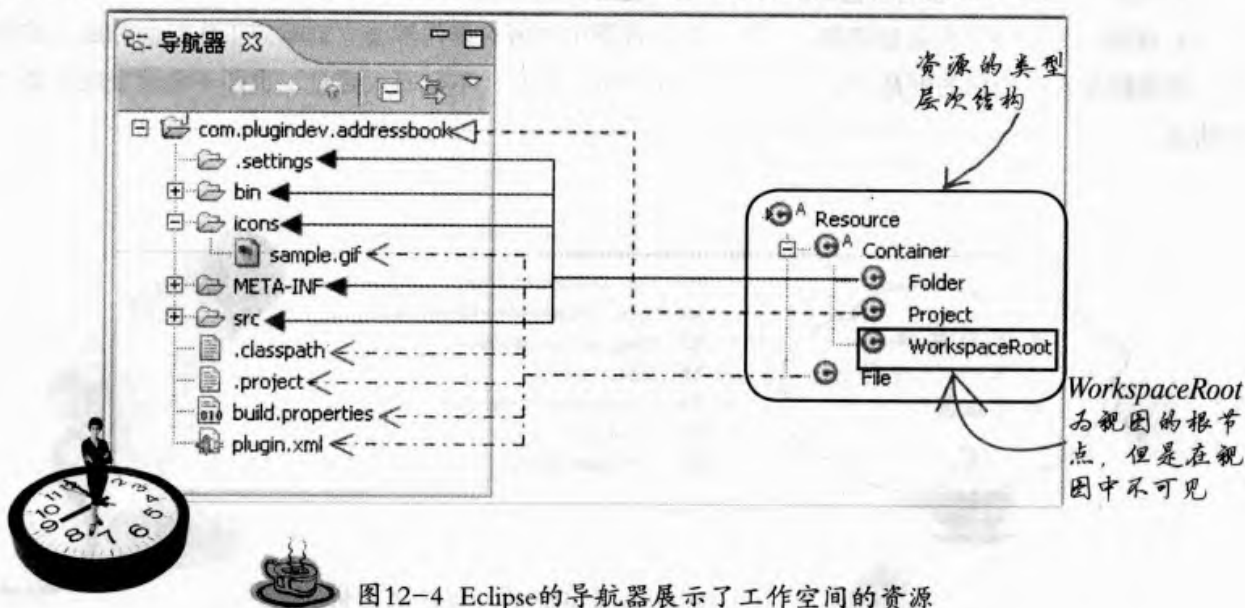


图12-4 Eclipse的导航器展示了工作空间的资源

表12-1中接口的方法可以被用来设置导航器。

表12-1 定制导航器需要使用的方法列表

方法名称	方法描述
getFrameList()	提供一个FrameList实例，这个实例记录了导航的历史信息。例如当需要导航到先前的资源视图时，可以调用FrameList的back()方法
getPatternFilter()	获得当前活跃的过滤器。过滤器中记录了导航器中不显示的资源（但该类资源有可能存在）
getSorter()	获得当前活跃的排序器。排序器指定了资源在导航器中显示的顺序
getViewer ()	获得当前的树查看器，导航器使用树查看器来显示资源
getWorkingSet()	获得当前的活跃的工作集。工作集中保存了当前可用的所有资源
setFiltersPreference()	设置新的Filter样式
setSorter()	设置新的资源排序器来调整排序策略
setWorkingSet()	设置新的工作集

ResourceNavigator是Eclipse中很重要的视图，Java包浏览器等许多其他的导航器都基于ResourceNavigator并且使用上面的这些方法来应用它们自己的配置。

12.2.2 PageBook视图

抽象类PageBookView是AbstractDebugView，ContentOutline和PropertyView父类，它支持视图显示某个工作台组件（IWorkbenchPart，如图12-1所示）的状态信息，如可以用它们来显示当前编辑器的状态信息。从名字中可以看出，PageBookView实例可以包含多个页面，标准页面一般显示当前活跃的工作台组件的状态信息，其他页面可以显示工作台其他组件的信息。可以在PageBookView的基础上来实现自定义的视图。

PageBookView的子类必须实现如表12-2所示方法。

表12-2 PageBookView的子类需要实现的方法列表

方法名称	方法描述
createDefaultPage()	需要在这个类中构建默认的页面。在当前工作台组件中没有特定的PageBookView页面打开时，这个默认的页面会出现
getBootstrapPart()	这个方法确定可用的工作台组件。重写这个方法后，用户可以确定当前可用的其他工作台组件（不同于当前正在使用的组件）
isImportant()	当需要针对某个特定的工作台组件构建出PageBookView页面时，这个方法必须设定为真值（True）
doCreatePage()	使用此方法构建PageBookView页面。只有当isImportant()方法的值为真时，这个方法才可以执行
doDestroyPage()	除去PageBookView页面

PageBookView的子类还可以重写更多的方法，如partActivated(),partBroughtToTop(),partClosed(),partClosed(),partDeactivated(),partOpened()等，这样就可以定义自己的页面顺序和页面内容了。

12.2.3 大纲视图

ContentOutline类用来显示编辑器内容的大纲，如图12-5所示，Eclipse预定义的大纲视图便是由此类定义的。它仅有的实例是由工作台创建并管理的，既无法被再次实例化也无法被继承。



图12-5 大纲视图显示了Java编辑器中类的内容



这个唯一的实例可以通过调用IWorkbenchPage的showView()方法显示，如下面代码所示。

```
page.showView("org.eclipse.ui.views.ContentOutline");
```

showView()方法的参数是视图的标识(下同)。如果这个工作台页面是当前活跃的面，则可以使用下面方法来获得它。

```
page = PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage();
```

编辑器如果想要同大纲视图相关联，需要提供合适的适配器(见第13章“编辑器”)，当下面的方法被调用时，大纲视图将获得大纲页面的实例。

```
editor.getAdapter(IContentOutlinePage.class);
```

12.2.4 属性视图

Eclipse的属性视图同大纲视图一样，无法被再次实例化或拥有子类，只能由工作台提供。但是，属性视图不仅针对编辑器，它还可以支持其他工作台部分。如图12-6所示，当在“包资源管理器”中选择Activator.java文件时，属性视图就会显示这个文件的更多细节信息。

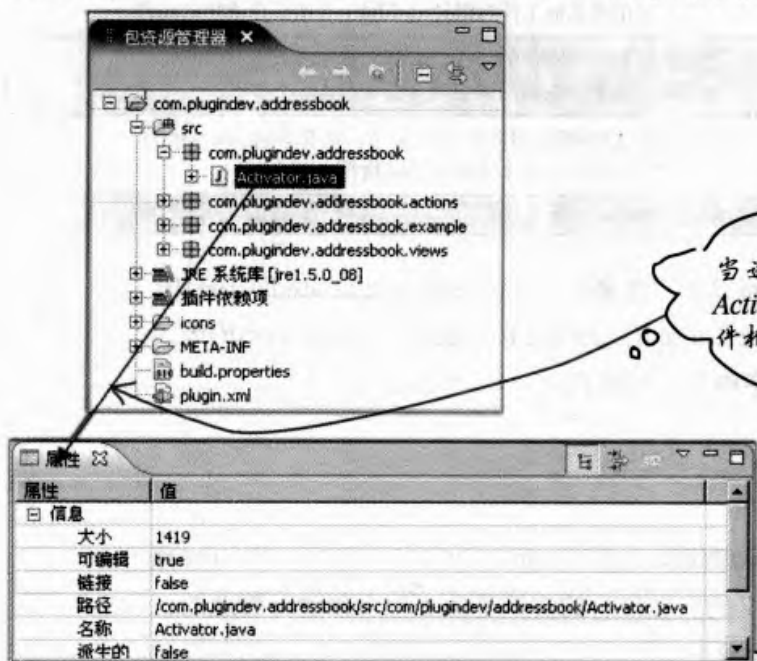


图12-6 属性视图显示了文件Activator.java的属性信息

当选中“包资源管理器”中的Activator.java时，属性视图显示文件相应的信息

属性视图可以显示和编辑特定对象的属性，可以使用IWorkbenchPage的showView()方法显示该视图，代码如下所示。

```
page.showView( org.eclipse.ui.views.PropertySheet );
```

当属性视图发现某个组件被激活时，它就会询问那个组件，看其是否能够提供一个PropertySheetPage实例，如果得到肯定的回答，则这个页面会包含在属性视图中。如果某个工作台部分希望属性视图能够响应它内部的变化，则它需要提供合适的适配器。当下面的方法被调用时，属性视图将会获得属性页面的实例。

```
part.getAdapter(IPropertySheet Page.dass);
```

12.2.5 任务视图和书签视图

任务视图和书签视图很相似，都是Eclipse提供的基本视图，在“窗口”→“显示视图”→“其他”的“常规”类别里可以找到这两个视图，它们用来增加新的任务和书签。这两个视图也无法被再次实例化或继承，它们是工作台提供的功能。图12-7中创建了Activator类中创建的5个书签，并显示了书签项同代码位置的对应关系。



图12-7 书签与代码间的对应关系

可以使用IWorkbenchPage的方法来显示它们，代码如下所示。

```
page.showView( org.eclipse.ui.views.BookmarkNavigator );
page.showView( org.eclipse.ui.views.TaskList );
```

可以在Eclipse编辑器左侧单击右键来在特定的位置添加任务或书签，打开这两个视图后，双击所选中的项，则Eclipse会跳转到所记录的位置。

注意：

上面的4个视图虽然不能实例化或继承，但可以被引用。在本书后面生成RCP程序的介绍中，将会使该视图成为地址本应用程序的一部分。



12.2.6 项目资源管理器

项目资源管理器使用CommonNavigator类来管理项目的资源，JDT针对Java元素创建了对应的资源管理器。当然，开发者也可以使用项目资源管理器为自己的应用程序提供树形结构。在基于EMF的应用程序中，通常使用项目资源管理器来配置树形结构，显示同业务相关的特定模型。

Eclipse 3.2新增了org.eclipse.ui.navigatorContent扩展点，为配置CommonNavigator的扩展提供了强大的支持。关于navigatorContent的更多内容，请参看Eclipse的在线帮助文档。

12.3 创建一个视图

在第10章已经通过使用Eclipse提供的模板为地址本插件创建了一个地址本视图，如果要手动创建视图应该如何做呢？创建视图同添加操作一样，也需要在清单文件plugin.xml中定义一些信息，并提供相应的代码。

创建一个视图需要如下三步。

- ❶ 在插件清单中定义视图类别；
- ❷ 在插件清单文件中定义视图；
- ❸ 创建包含代码的视图部分。

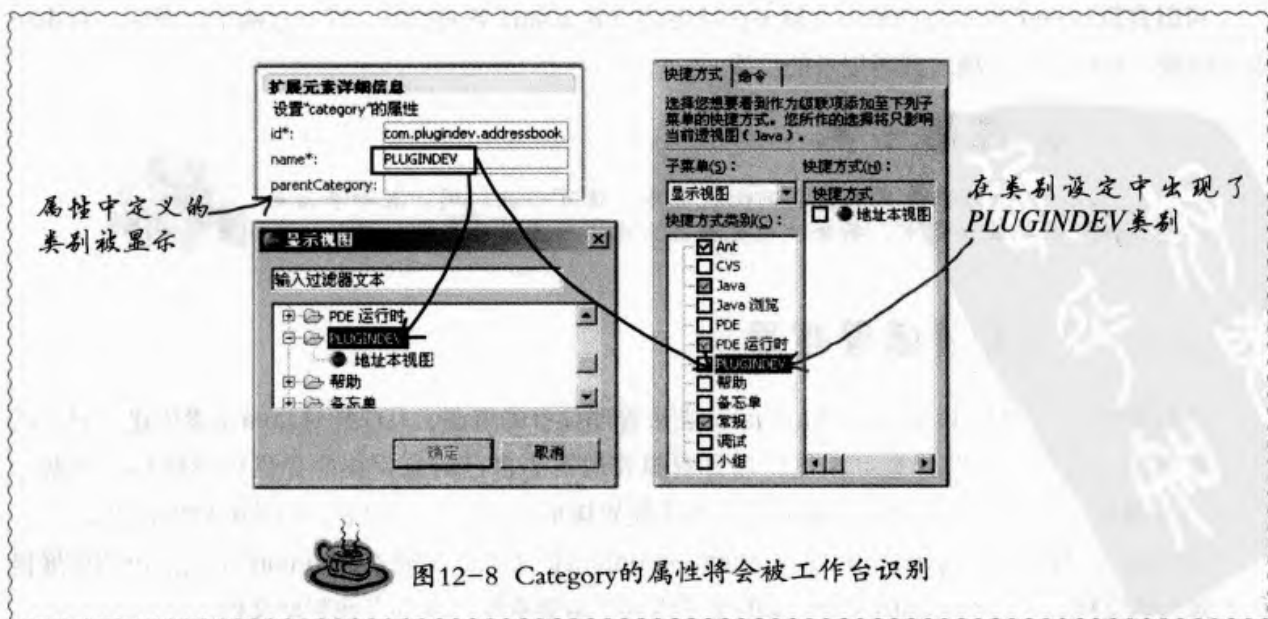
12.3.1 添加category

首先，为了定义新的视图类别（category），编辑插件清单，并导航到“扩展”页。如果没有显示org.eclipse.ui.views扩展点，则在右键上下文菜单中添加该扩展点，然后在右键上下文菜单中选择“新建”→“category”。category元素的属性如下所示。

- ★id*——类别的唯一标识符。
- ★name*——出现在“显示视图”对话框中的易读类名。
- ★parentCategory——父类别，引用其他类别的id。

第10章已经由模板创建了这个扩展点，并创建了PLUGINDEV类别，其属性如图12-8所示。类别创建之后，如果运行这个程序，在“窗口”→“定制透视图”的“快捷方式”选项卡中，便可以看到已经定义了的类别，同时，如果定义了视图，该类别也会出现在“显示视图”对话框中。

第10章已经由示例创建了这个扩展点，因此，直接选中该扩展点，在右键上下文菜单中选择“新建”→“category”，将该category的属性设置如图12-8所示。



12.3.2 在plugin.xml中声明视图

定义了视图类别之后，再次用右键单击“扩展”页上的org.eclipse.ui.views扩展，选择“新建”->“视图”命令来定义一个新的视图。同添加category一样，在“扩展元素详细信息”区域编辑视图的属性。第10章中已经定义了地址本视图，因此，现在直接选择“com.plugindev.addressbook.views.AddressView”，在右侧查看“地址本视图”的属性，如图12-9所示。

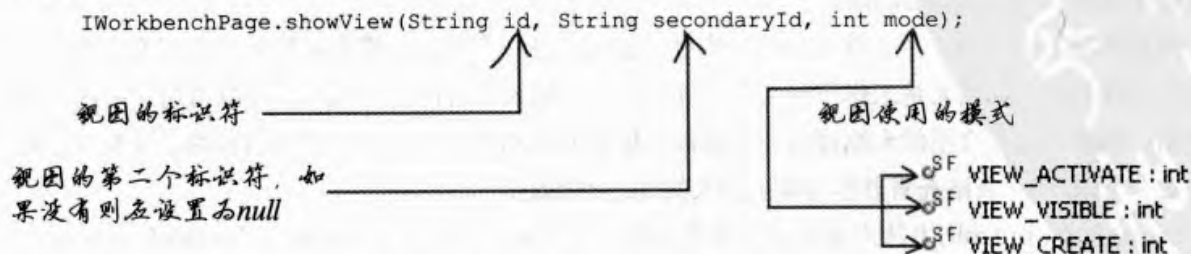


图12-9 “地址本视图”的属性



这些属性的含义分别如下所示。

- ★**id**——视图标识符，可以使用此标识符引用该视图。
- ★**name**——视图显示的名称，最终用户通过此名称来识别该视图。
- ★**class**——定义视图的类的完全限定名，该类必须实现org.eclipse.ui.IViewPart接口。一般的视图类都会继承org.eclipse.ui.ViewPart。
- ★**category**——该视图所属的类别，在其中输入视图类别的标识符。
- ★**icon**——与视图相绑定的图标，将会显示在视图的左上角和“显示视图”对话框中。图标的路径是安装插件的相对路径。
- ★**fastViewWidthRatio**——定义视图出现在工作台窗口时的宽度百分比。需要输入的是一个从0.5到0.95的浮点数。若未设定浮点值，Eclipse将使用默认的浮点值。
- ★**allowMuntiple**——定义是否允许在工作台页面中出现多个该视图的实例。如果允许的话，可以使用下面的方法来创建第二个实例。默认值为false。



视图使用的模式可以有三种选择，VIEW_ACTIVATE，VIEW_VISIBLE，VIEW_CREATE。它们的含义如表12-3所示。

表12-3 视图可以使用的模式

模式	描述
VIEW_ACTIVATE	激活定义视图的插件并在当前页面中显示视图
VIEW_VISIBLE	表示该视图可见, 但并不会立即在当前页面中显示
VIEW_CREATE	创建视图, 但只有包含该视图的位置文件夹, 未包含其他可见视图时, 该视图才会在页面中显示

12.4 视图类

视图的行为由一个实现了org.eclipse.ui.IViewPart接口的类定义。第10章介绍了Eclipse的视图模板创建出来的简单地址本插件视图, 本节将回顾第10章创建的视图代码, 并在其基础上扩充内容。

注意

学习本节需要首先对SWT和JFace机制有一些了解, 查看更多内容请看第7章。



12.4.1 视图方法

打开地址本插件中的com.plugindev.addressbook.views.AddressView类, 可以看到, 除了构造方法以外, 有以下两个方法。

★createPartControl(Composite)——该方法用来定义视图的用户界面, 是视图类方法中最重要方法 (参见第12.4.2节)。

★setFocus()——这也是一个必须实现的方法, 它将焦点设置到视图中合适的控件上。


除此之外, 如果视图中使用了图像或其他资源, 还应该使用dispose()方法来销毁这些资源。在后续的几节中, 会涉及视图类中的其他方法。

12.4.2 视图控制

Eclipse的视图可以包含任意类型和数量的空间, 但为了结构的方便, 通常在视图中仅包含一个单一的JFace表或树查看器控件 (参见第7章7.1节对查看器的介绍)。当然, 在视图中直接使用SWT控件或使用多个查看器也未尝不可, 但是尽管在视图中允许包含这些内容, 作为良好的设计习惯, 应该尽可能保证视图有一个简洁的结构。

视图的输入可以是一个工具专用的域模型, 也可以是一个特定的模型对象和与它相关的内容。输入可以是固定的, 如输入是工作空间中的“导航器”; 输入也可以是动态的, 可以由用户所选的内容来控制, 在第12.2节介绍的大纲视图和属性视图便是使用动态输入来决定视图内容的。从本节开始, 将定义一个以JFace表格查看器作为单一输入的地址本视图。

createPartControl()方法用来定义视图界面的各个组成元素。图12-10显示了createPartControl()的结构。

在  处, Eclipse模板创建了一个表格查看器 (org.eclipse.jface.viewers.TableViewer), 并

且规定了表格查看器的显示样式，每种方式描述以“|”来分隔。在此，为其添加一个新的显示属性 SWT.FULL_SELECTION，使得当用户作出选择时，整个行会被突出显示，代码如下所示。

```
viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL |  
SWT.V_SCROLL | SWT.FULL_SELECTION);
```

还需要定义表格的显示方式。表格的第一列记录的是“姓名”，第二列记录的是“类别”，然后为每个列定义了相应的图像、宽度。最后，要让表头在视图中显示出来。

● 中包含的代码用来对查看器进行定制，包括设置输入模型（参见第12.4.3节），设置查看器的内容提供者（参见第12.4.4节），设置查看器的标签提供者（参见第12.4.5节）。

● 中包含的代码用来定义视图的行为。这些行为包括视图的通用行为，视图上下文菜单中包括的操作，对双击事件的响应，视图工具栏中定义的操作等。

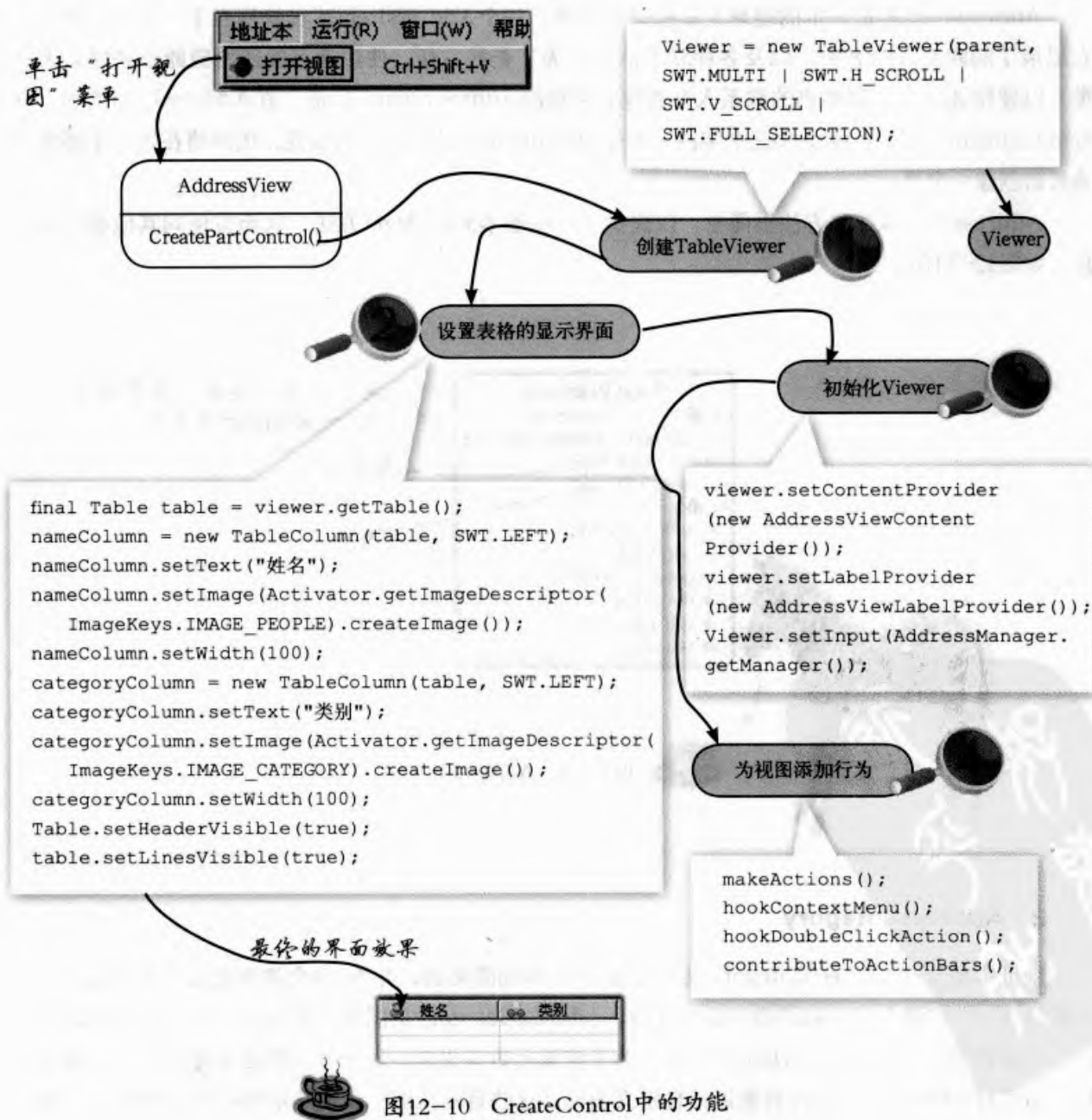


图12-10 CreateControl中的功能

12.4.3 视图模型

为了实现地址本视图的功能，本节将创建视图使用的模型。这些模型将作为viewer.SetInput()中的参数被表格查看器使用。

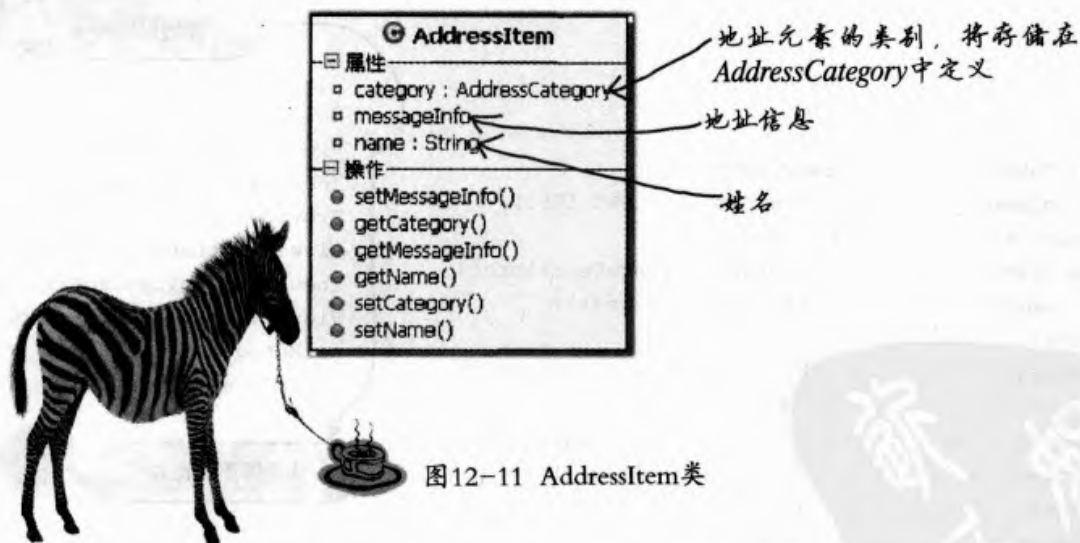
要创建的模型如下所示。

- ★AddressItem——地址元素模型对象。
- ★AddressCategory——地址元素的类别。
- ★AddressManager——管理Address模型对象。

1. AddressItem

AddressItem类是基本的地址元素模型对象类，一个AddressItem其实就相当于一个电子名片，它记录了联系人的名字，以及各种联系信息。为了查找方便，使用者可能还希望能够将联系人分类，以便快速区分不同类别的联系人并查找到需要的AddressItem，因而，在AddressItem类中，有name、category和messageInfo三个字段。由于category本身还具有一些属性，因而将在地址本插件中为其新建一个类。

AddressItem本身的方法很简单，仅提供了一些基本的set和get方法，从而方便同其他类传递信息，如图12-11所示。



2. AddressCategory

地址本插件在AddressCategory类中定义了9个不同的类别，并为这9个类别定义了优先级。为了对第12.4.6节将要介绍的视图排序提供支持，AddressCategory类需要实现java.lang.Comparable接口，重写其中的compareTo(Object)方法。为了使显示效果更佳，地址本插件还为每个类别定制了图标。为了对AddressCategory对象返回自定义图标（这些自定义图标在本章示例源代码中的icons文件

夹中可以找到），可以在插件的生命周期内缓存这些图标，并在插件关闭时处理它们（参见第12.9节“加载和卸载图标”）。

AddressCategory是一个抽象类，其中的getImage()方法是未实现的，这是为了方便对不同的地址类别使用不同的图像，并能使用一个统一的外部接口。其他对象仅需要调用getImage()方法就可以了，该方法能够根据具体的类型选择正确的图像。

AddressCategory的类图见图12-12所示，在该图中还列出了AddressCategory中定义的9个类别对象。

创建一个AddressCategory实例需要提供类名和优先级两个参数。AddressCategory的构造方法如下所示。

```
public AddressCategory( String categoryName, int priority)
{
    this.categoryName = categoryName;
    this.priority = priority;
}
```

*priority*用来设置优先级，以后的排序操作将根据优先级来排序

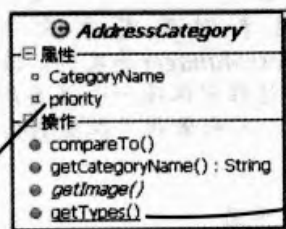
在创建类别实例的同时实现了getImage()方法，将不同的图像绑定到不同的类别上，如UNKNOWN实例的定义如下所示。

```
public static final AddressCategory UNKNOWN =
    new AddressCategory( "未分类", 0)
{
    public Image getImage()
    {
        return ImageKeys.
            getImageDescriptor(ImageKeys.IMG_CAT_UNKNOWN).createImage();
    }
};
```

ImageKeys是一个终态类，它里面保存了所有的图像元素的路径，并且提供访问图像描述符的方法（关于ImageKeys和图像描述符的更多信息，参见第12.9节“加载和卸载图标”）。

比较方法compareTo()是在优先级的基础上进行的，它的实现方式如下所示。

```
public int compareTo(Object obj){
    return this.priority
        - ((AddressCategory) obj).priority;
}
```



AddressCategory [] TYPES

地址类别对象的类型集合，存储了所有的预定义的类型对象

共设置了9个优先级，8为最高优先级

优先级	类别对象	含义	表示图像
0	UNKNOWN,	未分类	
1	ORDINARY,	普通	
2	MATE,	同事	
3	BUSINESS,	商业伙伴	
4	FRIENDS,	朋友	
5	FAMILY,	家庭	
6	VIP,	VIP	
7	TEACHER,	师长	
8	LOVER,	伴侣	



图12-12 AddressCategory的类图以及该类中定义的9个类别对象

3. AddressManager

地址本视图应该显示AddressItem对象的同一个集合，因此，AddressManager是一个负责维护此全局集合的单态类。因为可能会有多个对象同时访问AddressManager，所以管理器必须能够在信息更改时，通知注册的监听器。

为了从不同的类中访问AddressManager，该类的getManager()方法实现代码如下所示。

```
public static AddressManager getManager()
{
    if(manager == null)
        manager = new AddressManager();
    return manager;
}
```

当第一次调用该方法时，便会创建出工作空间中唯一的manager实例，在以后调用时仅简单返回该实例。

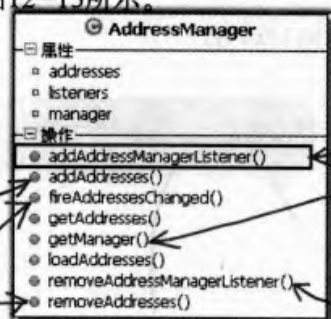
※ 注意 ※

这样处理也会增加潜在的隐患，可能不同的线程会同时访问manager对象，但是，由于只能够从UI线程访问FavoritesManager，所以不需要担心线程的危险性。“向导和对话框”一章涉及了更多从其他线程进入UI线程的内容。

AddressManager的类图如图12-13所示。

添加和删除模型，如果不希望其他类使用这两个方法直接操作模型，将此两个方法设置为protected类型

响应模型更改



注册监听器，这样当监听到修改模型事件发生时，就可以做出响应了

此方法是一个静态方法，采用单例模式来获得AddressManager的实例。在这过程中仅有一个该类的实例，从而保护了模型数据

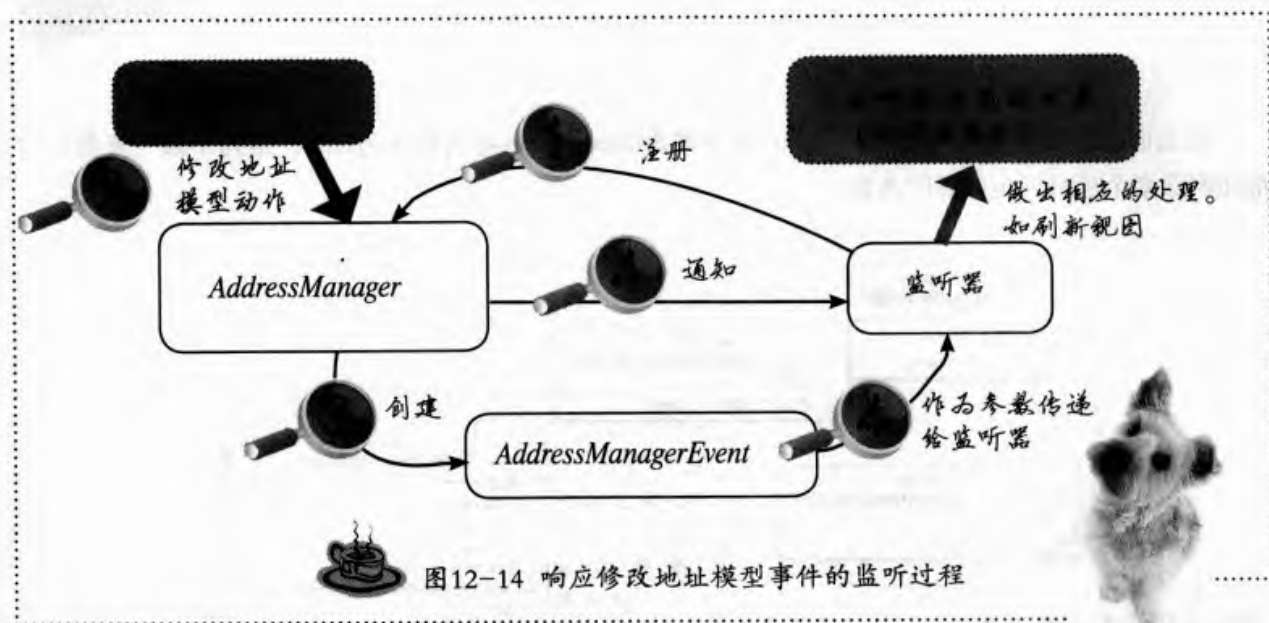
移除监听器



图12-13 AddressManager的类图

以后将增强AddressManager，用来持久保存Eclipse会话间的列表（参见第12.8节“视图状态”），但在目前，为了查看视图的显示效果，临时用固定的内容代替。初始化视图的输入在私有方法loadAddresses()中实现。loadAddresses()为每个类别创建了一个地址元素，并将它们放入地址元素的集合addresses中。

为了提供对模型的监听，AddressManager维护了一个监听列表，需要同AddressItem模型打交道的对象需要在AddressManager中注册监听器，以便对AddressManager实例发送请求。对监听器的管理是通过addAddressManagerListener()和removeAddressManager()实现的。注册了监听器后，如果模型需要更改，fireAddressesChanged()方法便开始执行。监听的过程如图12-14所示。



首先，监听器在AddressManager中注册自己，等待AddressManager的通知；当修改地址模型动作抵达AddressManager后，AddressManager就创建AddressManagerEvent的事件（该事件中包含了这一动作要修改的模型），并将这个事件作为参数传递给监听器；监听器在接到这一事件通知后，调用相应的方法来改变作用的对象。

这种监听方式的好处是，AddressManager不需要知道是什么对象发出的修改地址模型的动作，也不需要知道将要影响什么对象，只需要管理监听器的注册和通知就可以了。这遵循了面向对象设计中的“每个类为自己负责”的设计理念。

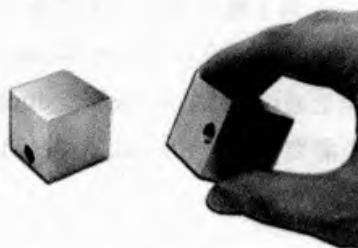
12.4.4 视图内容

创建了模型对象后，就需要将它们链接到视图，结构化内容提供者（IStructuredContentProvider）负责从输入对象中提取对象，并把它们传递给表格查看器进行显示。接下来，在AddressView自动生成的内部类的基础上，创建名为AddressesViewContentProvider的对象。该对象将使用AddressManager作为输入对象，表格的一行对应一个AddressItem元素。

AddressesViewContentProvider同AddressView自动生成的内部类还有一个区别，它实现了

AddressManagerListener接口。由于地址本插件需要在模型改变时，更新相应的显示，这就要求AddressViewContentProvider监听模型的变化。AddressViewContentProvider的类图结构如图12-15所示。AddressView类对AddressManagerListener中的addressesChanged()方法的实现代码如下所示。

```
public void addressesChanged(AddressManagerEvent event) {
    // TODO 自动生成方法存根
    viewer.getTable().setRedraw(false);
    try{
        viewer.remove(event.getItemsRemoved());
        viewer.add(event.getItemsAdded());
    }
    finally{
        viewer.getTable().setRedraw(true);
    }
}
```



在上面的方法中，刚开始将setRedraw设置为false，最后将其恢复为true，是为了减少查看器增加和删除多个项时发生闪烁的次数。

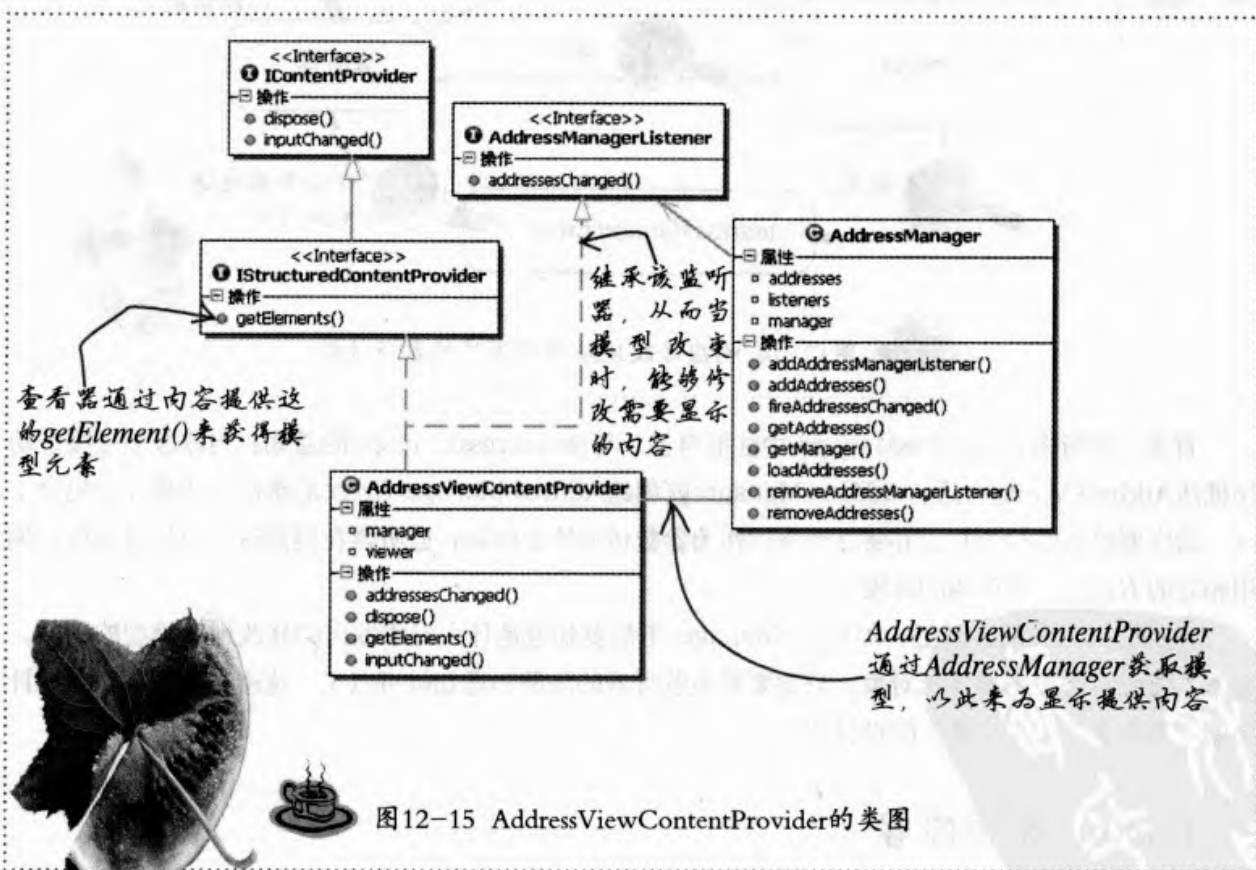


图12-15 AddressViewContentProvider的类图

在inputChanged()方法中，涉及了输入的改变，当输入改变时，AddressManager实例要卸载旧的输入在其上注册的监听器，并加载新的实例在其上注册的监听器，代码如下所示。

```
public void inputChanged(Viewer viewer, Object oldInput, Object newInput) {

    this.viewer = (TableViewer)viewer;
    if(manager != null)
```

```

manager.removeAddressManagerListener(this);
manager = (AddressManager)newInput;
if(manager != null)
{
    manager.addAddressManagerListener(this);
}
    
```

除去旧的监听器

添加新的监听器

12.4.5 视图标签

标签提供者 (LableProvider) 将获取内容提供者返回的一个表格行对象，并提取要显示到列上的值。由于模板已经在 AddressView 中创建了标签提供者内部类，因而只要将其重构为一个顶级类并对从新建的对象模型中提取的值进行改写就可以了。

视图标签类的实现比较简单，只需要根据对表格列的判断来显示相应区域的文本和图像即可。

最后，在 CreatePartControl() 代码的“初始化viewer”部分更改相应的参数，创建的模型就链接到了视图上，一个最基本的地址本视图就做好了。地址本视图完成的效果如图12-16所示。



12.4.6 视图排序

查看器排序器 (ViewerSorter) 用于对内容提供者提供的元素进行排序。当没有使用查看器排序器时，查看器将会按照内容提供者返回的顺序显示元素（默认顺序如图12-16所示）。当内容提供者提供行对象时，ViewerSorter应该在显示这些行对象之前负责排序行对象。在“地址本视图”中，每个地址元素项可以以姓名和类别两个排序准则进行升序或降序排序。

AddressViewerSorter将排序委托给两个比较器来完成，每个比较器分别对应于上面列出的一个准则。另外AddressViewerSorter监听列标题上的鼠标单击事件，并根据选择的列重新排列内容。再次单击列可以反转排序顺序。

为了对排序器提供支持, 需要在AddressView中增加创建排序器的相应代码, 如下所示。

```
private AddressViewerSorter sorter;
public void createPartControl(Composite parent) {
    .....
    createTableSorter();
    .....
}

private void createTableSorter() {
    Comparator nameComparator = new Comparator() {
        public int compare(Object o1, Object o2) {
            return ((AddressItem) o1).getName()
                .compareTo(((AddressItem) o2).getName());
        }
    };
    Comparator categoryComparator = new Comparator() {
        public int compare(Object o1, Object o2) {
            return ((AddressItem) o1).getCategory()
                .compareTo(((AddressItem) o2).getCategory());
        }
    };
    sorter = new AddressViewerSorter(
        viewer, new TableColumn[] { nameColumn, categoryColumn },
        new Comparator[] { nameComparator, categoryComparator });
    viewer.setSorter(sorter);
}
```

排序器中最重要的就是比较器, 在此创建基于内容和类别的两个比较器

创建排序器, 在其中将表格列和比较器绑定



排序器创建和排序的过程如图12-17所示。AddressView在其createPartControl()方法中创建AddressViewerSorter排序器, 该排序器在其构造方法中创建了对表格表头的监听器, 监听表格单击事件的发生。表格单击事件将会触发监听器对表格内容排序, 并根据排序后的结果刷新AddressView视图。

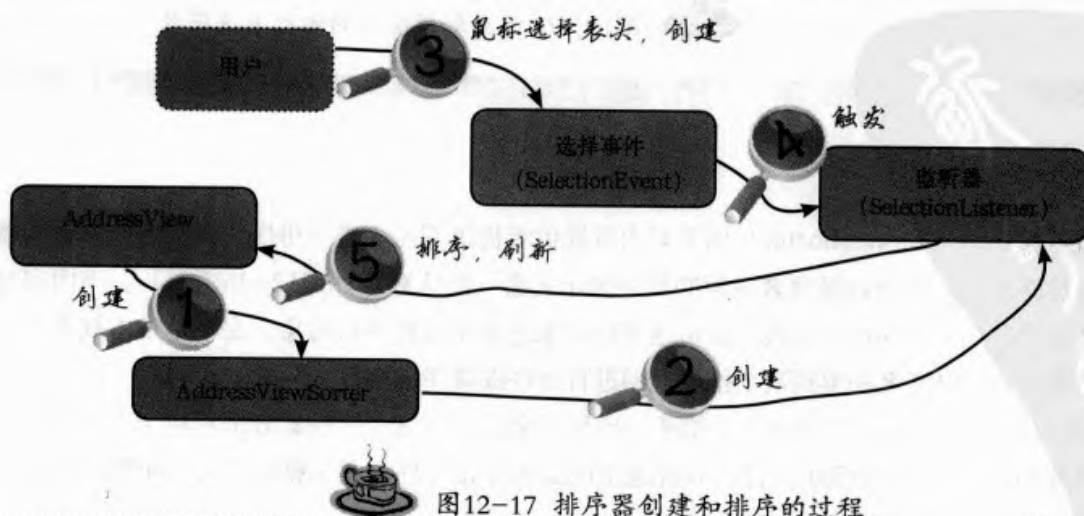


图12-17 排序器创建和排序的过程

图12-18显示了按类别排序的效果。

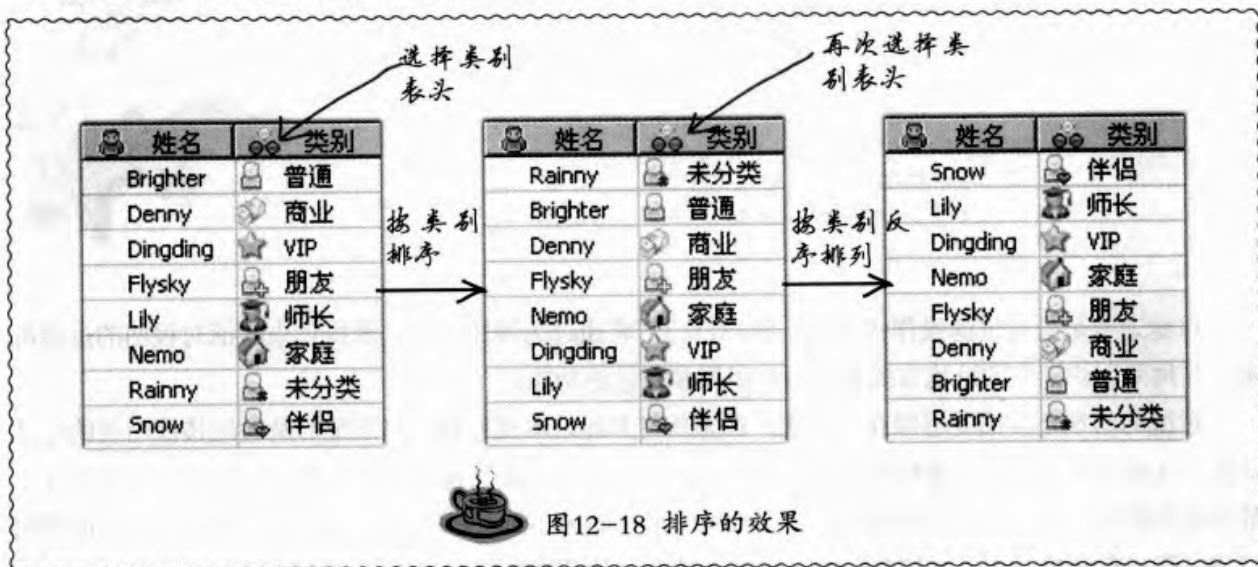


图12-18 排序的效果

排序的结果还应该能够保存，使用户在重新打开地址本视图时仍然出现的是关闭该视图前的排序状态，从而为用户提供一致性操作。在现阶段，排序还不支持保存，每次关闭视图后再次打开，都将看到图12-18中的左侧显示的默认视图。在后面的第12.8.1节“存储排序和过滤信息”中，将会给出完善排序器的方法，使其增加记忆功能。

12.4.7 视图过滤

ViewerFilter子类用于确定显示内容提供者返回的哪些行对象，不显示哪些行对象。一个视图仅可以有一个内容提供者、一个标签提供者和一个排序器，但过滤器却可以拥有任意多个。当应用多个过滤器时，只有那些满足所有这些过滤器的项才能显示出来。

类似于第12.4.6节讨论的排序，地址本视图可以用以下两个过滤准则进行过滤。

★人名

★类别

这里使用了Eclipse的通配符过滤类org.eclipse.ui.intenal.misc.StringMatcher。这个类位于内部包中，必须先将其复制到com.plugindev.addressbook.util包。

AddressViewerCategoryFilter类为地址本元素的类别创建了一个过滤机制。一种用户希望的过滤方式是系统默认提供所有的地址项，当用户输入某个类别时，系统将过滤掉其余的类别，仅显示当前类别。如图12-19所示。

setPattern()方法实现的代码如下所示。

```
public void setPattern(String newPattern) {
    boolean filtering = matcher != null;
    if (newPattern != null && newPattern.trim().length() > 0) {
        pattern = newPattern;
        matcher = new StringMatcher(pattern, true, false);
        if (!filtering)
            viewer.addFilter(this);
    }
}
```



```

else
    viewer.refresh();
}
else {
    pattern = "";
    matcher = null;
    if (filtering)
        viewer.removeFilter(this);
}
}

```



过滤通常是通过过滤操作实现的。第12.5.3节将通过为视图添加过滤操作来激活对视图的过滤机制，并按照用户期望的过滤方式为地址本视图添加过滤功能。

过滤的结果还应该能够保存，使用户在重新打开地址本视图时仍然出现的是关闭该视图前的过滤状态，从而为用户提供一致性的操作。在现阶段，过滤还不能支持保存，每次关闭视图后再次打开，都将显示默认的视图（重新调用AddressManager中的loadAddresses()方法）。本章将在后面的第12.8.1节“存储排序和过滤信息”中，完善过滤器，使其增加记忆功能。

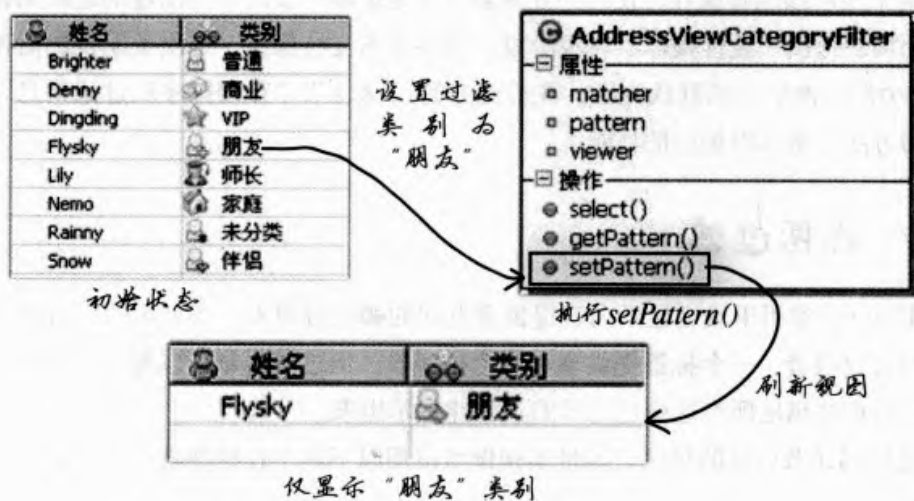


图12-19 期望的过滤方式

12.5 为视图添加操作

一个视图操作可以作为视图上下文菜单中的一个菜单项出现，也可以作为视图标题栏右侧的工具栏按钮出现，还可以作为视图下拉菜单中的一个菜单项出现。本节将以编程的方式为视图添加三个操作，添加地址元素、删除地址元素、过滤地址元素，并在稍后讨论为视图添加操作的更添一层次的话题。

12.5.1 视图选择

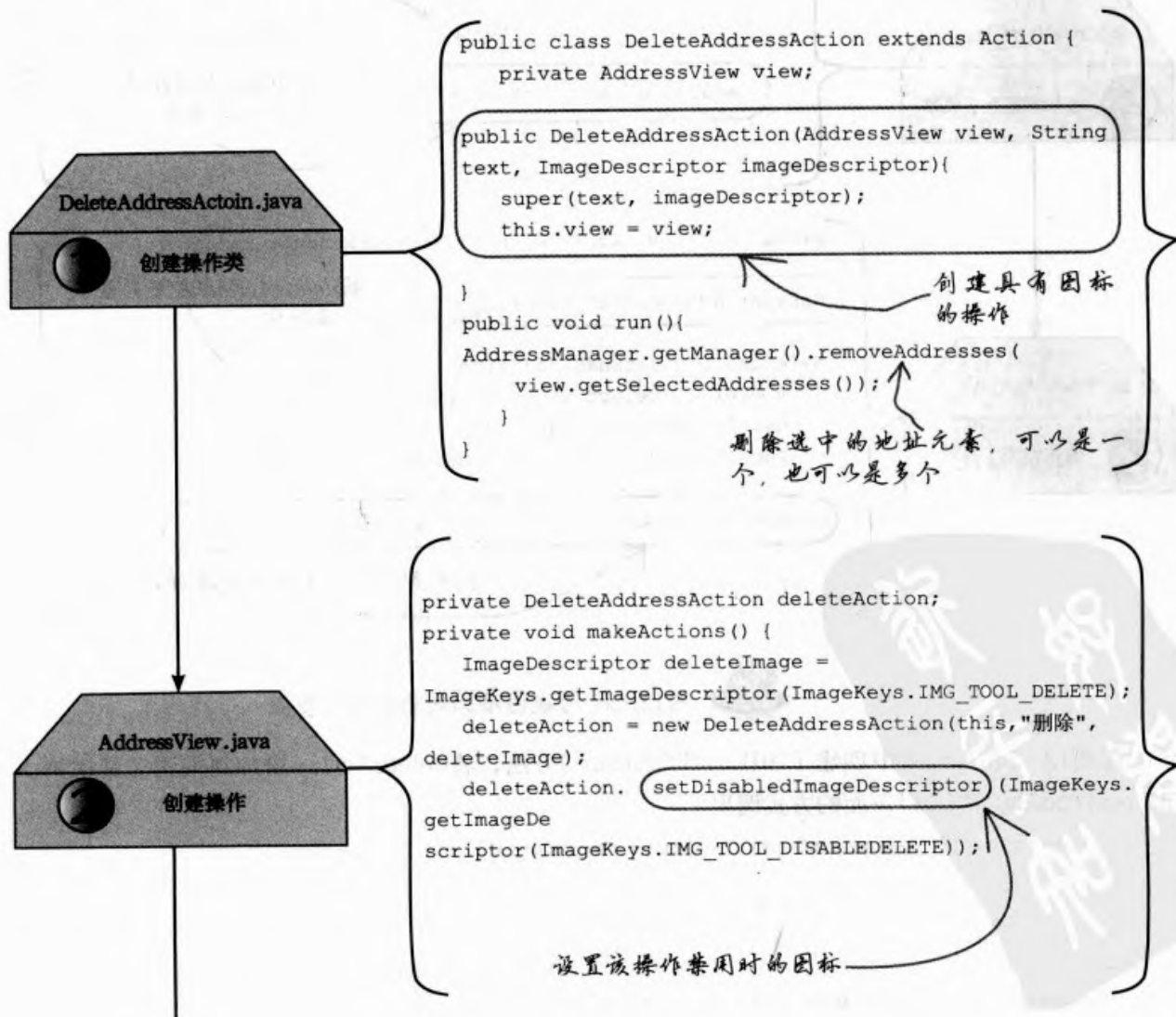
在为视图添加操作之前，还需要做一些准备工作。对象模型和视图控件均准备就绪后，视图的操

作将需要某种方法来确定当前选择了哪些AddressItem项。在AddressView中添加下面的方法，以便可以在选择的项上执行操作。

```
public AddressItem[] getSelectedAddresses()
{
    IStructuredSelection selection = (IStructuredSelection)viewer.getSelection();
    AddressItem[] items =
new AddressItem[ selection.size()];
    Iterator iter = selection.iterator();
    int index = 0;
    while(iter.hasNext())
        items[ index++] = (AddressItem)iter.next();
    return items;
}
```

12.5.2 添加/删除操作

用编程的方式添加/删除操作到视图上需要4个步骤，图12-20演示了在每一步需要编写的代码。



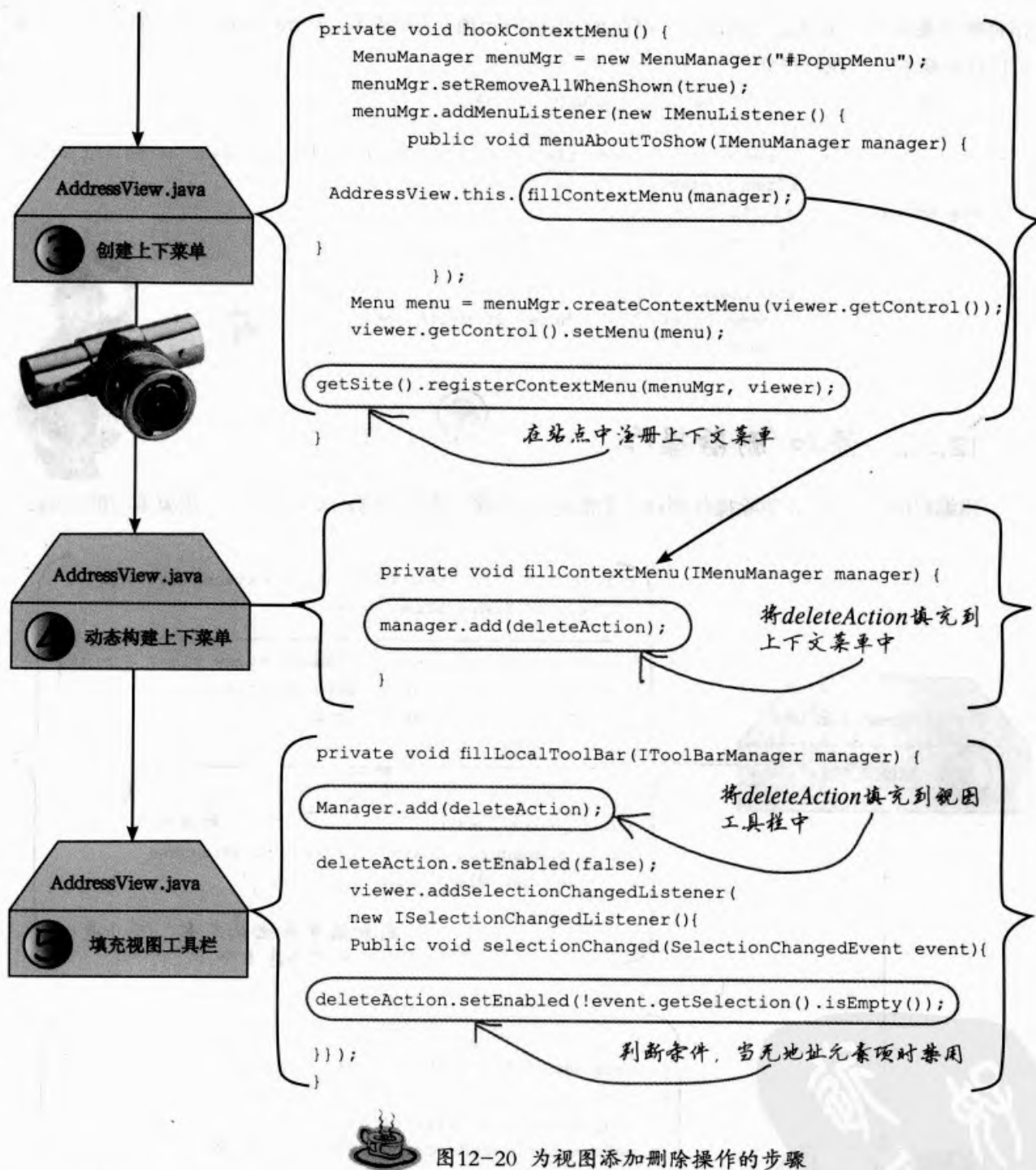


图12-20 为视图添加删除操作的步骤

在图12-20的第5步中创建了fillLocalToolBar()方法，将deleteAction增加到视图工具栏中。fillLocalToolBar()将被以下面的方式调用。

```
public void createPartControl(Composite parent) {
    .....
    contributeToActionBars();
}
.....
private void contributeToActionBars() {
```

```
IActionBars bars = getViewSite().getActionBars();
fillLocalToolBar(bars.getToolBarManager());
```

删除操作需要处理这样的情况，当地址本视图中没有地址元素时，删除操作应该不可用。因而，需要在第二步“创建操作”中，设置禁用该操作时的图标（灰色），代码如下所示。

```
deleteAction.setDisabledImageDescriptor(
    ImageKeys.getImageDescriptor(ImageKeys.IMG_TOOL_DISABLEDDELETE));
```

接下来，为视图增加SelectionChangeListener监听器来监听选择的改变，并且根据选择的内容设置删除操作的可用性。代码如下所示。

```
deleteAction.setEnabled(false);
viewer.addSelectionChangedListener(
    new ISelectionChangedListener(){
        public void selectionChanged(SelectionChangedEvent event){
            deleteAction.setEnabled(!event.getSelection().isEmpty());
        }
    });
}
```

为视图增加“添加”操作和增加“删除”操作的步骤相同，这里不再细述。添加完操作后，需要在AddressView的createPartControl()方法中引用相应的方法（本例中，Eclipse自动创建的模板已经加入进来）。这样，自定义的操作便可以运行了，运行的效果如图12-21所示。



12.5.3 过滤操作

本节用编程的方式为地址本视图增加过滤操作，以便启用和禁用名称过滤。这个操作同“添加”、“删除”操作类似，只不过它不是在上下文菜单中显示，也不是在工具栏按钮中显示，而是在视图工具栏中的下拉菜单中显示。该操作暂时将使用简单的InputDialog对话框来提示输入名称过滤的模式，在本书的后面将使用一个专门的“地址本”视图过滤对话框来代替该对话框（参见第15.4.2节“为地址本视图创建过滤器对话框”）。

图12-22演示了为视图添加过滤操作的过程。

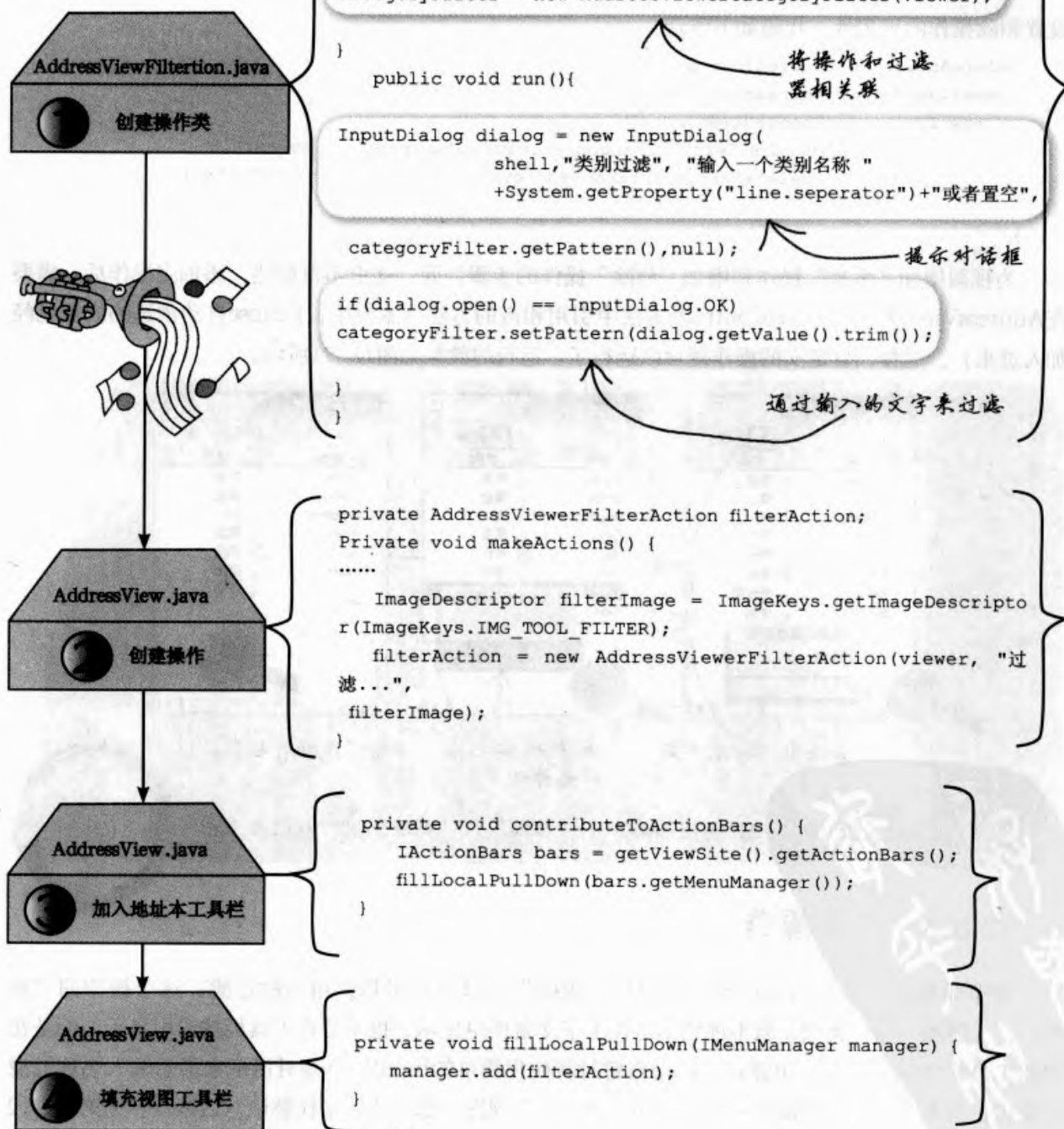


图12-22 为视图添加过滤操作的步骤

12.5.4 快捷键支持

本书第11.7节使用插件的方式为地址本插件增加了快捷键，本节将介绍以编程的方式为视图提供快捷键支持的方法。

以编程的方式为视图提供快捷键支持也非常简单。作为示例，这里为“删除”操作增加快捷键支持。为此，需要创建hookKeyboardAction()方法，并在createPartControl()中调用该方法。增加快捷键支持的步骤如图12-23所示。

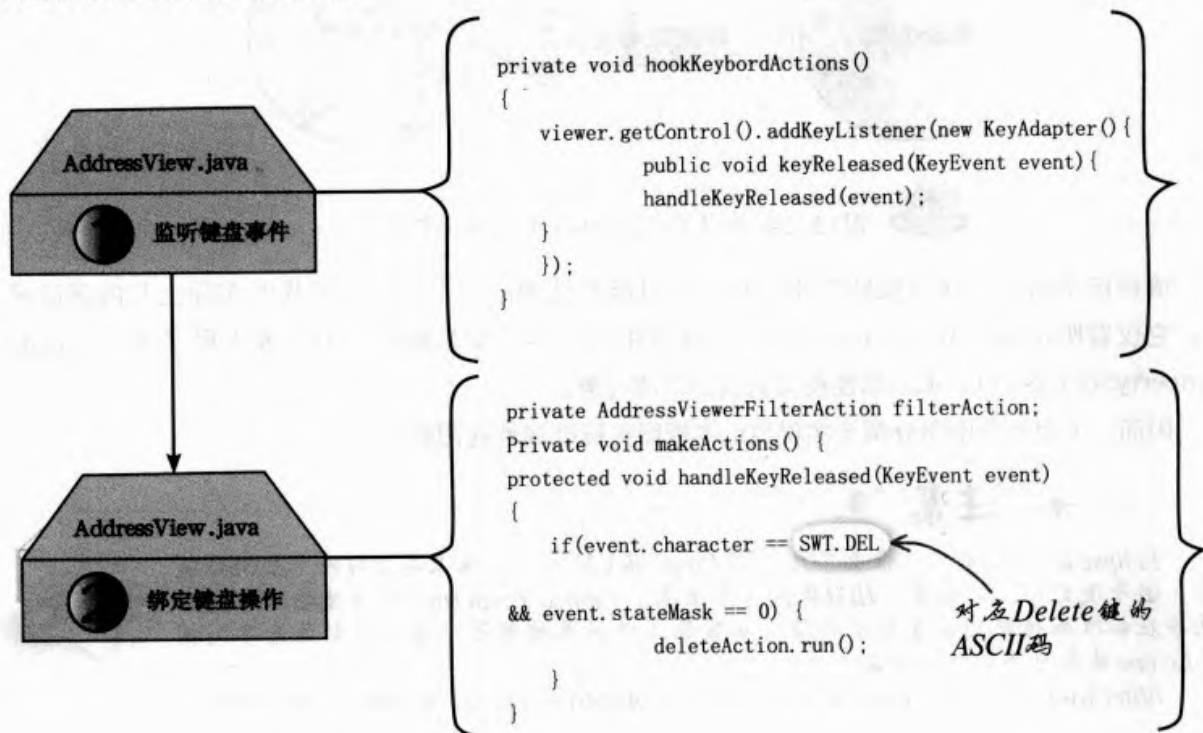


图12-23 为删除操作增加快捷键支持的步骤

12.6 视图间通信

视图在Eclipse用户界面中经常扮演支持性的角色。共享的视图（如Properties视图）可以通过用户快速访问追加的信息，对活动视图或编辑器起补充作用。为了说明多个Workbench部分如何协同工作来有效地把某些信息通知给用户，地址本插件增加了对标准属性视图的支持（有关对属性视图的介绍请参见第12.2.4节“属性视图”）。本节主要介绍地址本视图和属性视图间的通信方式，并在第12.6.4节简要介绍地址本视图监听其他视图的方式。

12.6.1 对属性视图提供支持

在介绍具体的实现方法之前，插件开发者需要明确地址本视图和属性视图的职责，具体如下所示。

★地址本视图共享它查看器中被选中的内容。

★属性视图侦听地址本查看器中被选中的内容，并决定是否请求被选中的对象。

地址本视图和属性视图关系的示意图如图12-24所示。

属性视图支持所有实现IResource接口的资源。在创建的地址本插件中，视图中显示的所有地址元素并没有实现IResource接口。

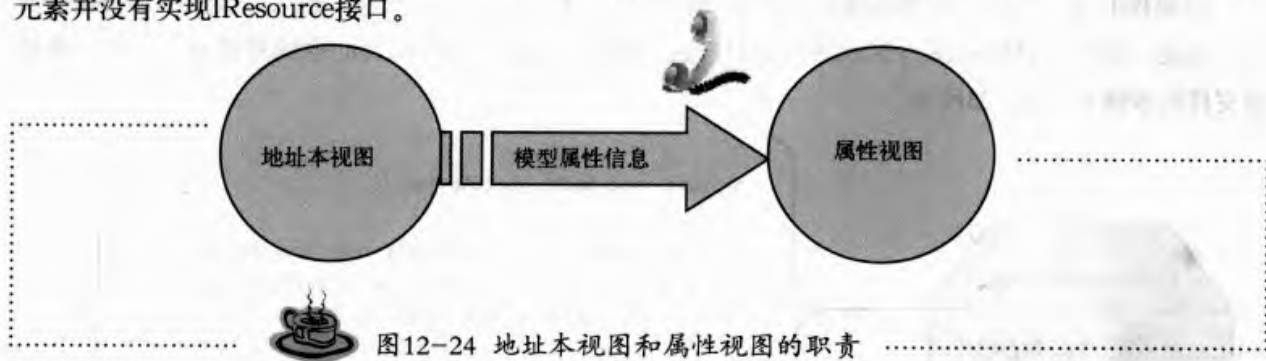


图12-24 地址本视图和属性视图的职责

值得庆幸的是，属性视图并不直接感知可能被选中的每个对象。该视图实际上与内部情况无关，它仅监听活动的Workbench部分中被选中的内容，如果被选中的内容实现了或可以适配于IPropertySource接口，那么属性视图就会请求该对象。

因而，下面两节中将分两步实现地址本视图对标准属性视图的支持。

※ 注意 ※

Eclipse在3.1版本后，推出了Tabbed Properties View，这原来是IBM的一个内部项目，由于项目开发的需要，IBM将其公开出来。Tabbed Properties View使插件开发者能够在标准属性视图的基础上创建更加复杂美观的属性视图，有关介绍请参阅相关Eclipse的学习资料，网址为：

http://www.eclipse.org/articles/Article-Tabbed-properties/tabbed_properties_view.html。



12.6.2 共享并监听地址本视图查看器

地址本视图的用户界面中使用了JFace的表格查看器，这种方式很容易将视图中被选中的内容共享。当前被选中的内容可以通过相应的内容提供者（ContentProvider）来共享，这种方式要求被选中的内容提供者必须实现ISelectionProvider接口，成为选择提供者。所有JFace的查看器都已经实现了ISelectionProvider接口（有关查看器的更多细节，请参阅第7章）。

为了在地址本视图中共享被选中的内容，必须调用WorkbenchSite，在视图类的createPartControl()中调用如下代码。

```
getViewSite().setSelectionProvider(viewer);
```

如果需要的话，可以使用自己实现ISelectionProvider接口的对象，在调用setSelectionProvider方法时，该对象作为它的参数被传递。

12.6.3 提供显示在Properties视图中的内容

AddressItem类直接继承了Object类，由于属性页面仅能够理解IPropertySource对象，它对

AddressItem无动于衷。

为了对属性视图提供支持，AddressItem需要实现IPropertySource接口。但是，可能在以后的需求中，开发人员希望AddressItem为更多的视图或其他对象提供支持，使用这种方式就必须实现越来越多的接口，这当然不是一个好主意！

Eclipse提供的org.eclipse.core.runtime.IAdaptable接口为开发者提供了解决方案。它允许一个对象把它不理解的类型对象转换成另外一种可以询问和控制的类型对象。这是对Adapter模式的经典使用。它意味着，在地址本视图中选择的项可以按属性视图的要求，翻译成IPropertySource元素，而对属性视图本身并不做任何修改。因而，只需要为AddressItem实现一个IAdaptable接口就可以满足所有的需求。

※ 注意 ※

Adapter模式是一种结构型模式，又名包装器Wrapper。许多时候，为复用而设计的工具箱类（如这里的IPropertySource）不能被复用的原因仅仅是因为它的接口与专业应用领域（如这里的AddressItem）所需要的接口不匹配。关于Adapter模式的更多内容请参见Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides所著的《设计模式：可复用面向对象软件的基础》一书。



为此，需要更改原来的AddressItem类，为其添加Adapter接口，并使用getAdapter()方法，其实现如下所示。

```
public class AddressItem implements IAdaptable{
    .....
    public Object getAdapter(Class adapter) {
        if(adapter == IPropertySource.class)
            return new AddressItemPropertySource(this);
        return null;
    }
}
```

为属性视图提供支持，返回属性视图可以识别的属性源

从上面的代码中可以看到，getAdapter()方法的调用者需要传入一个代表特定接口的Class对象，该方法则会返回一个具有该接口的对象，其示意图如图12-25所示。

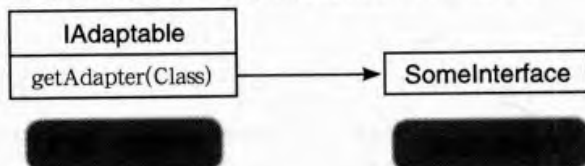


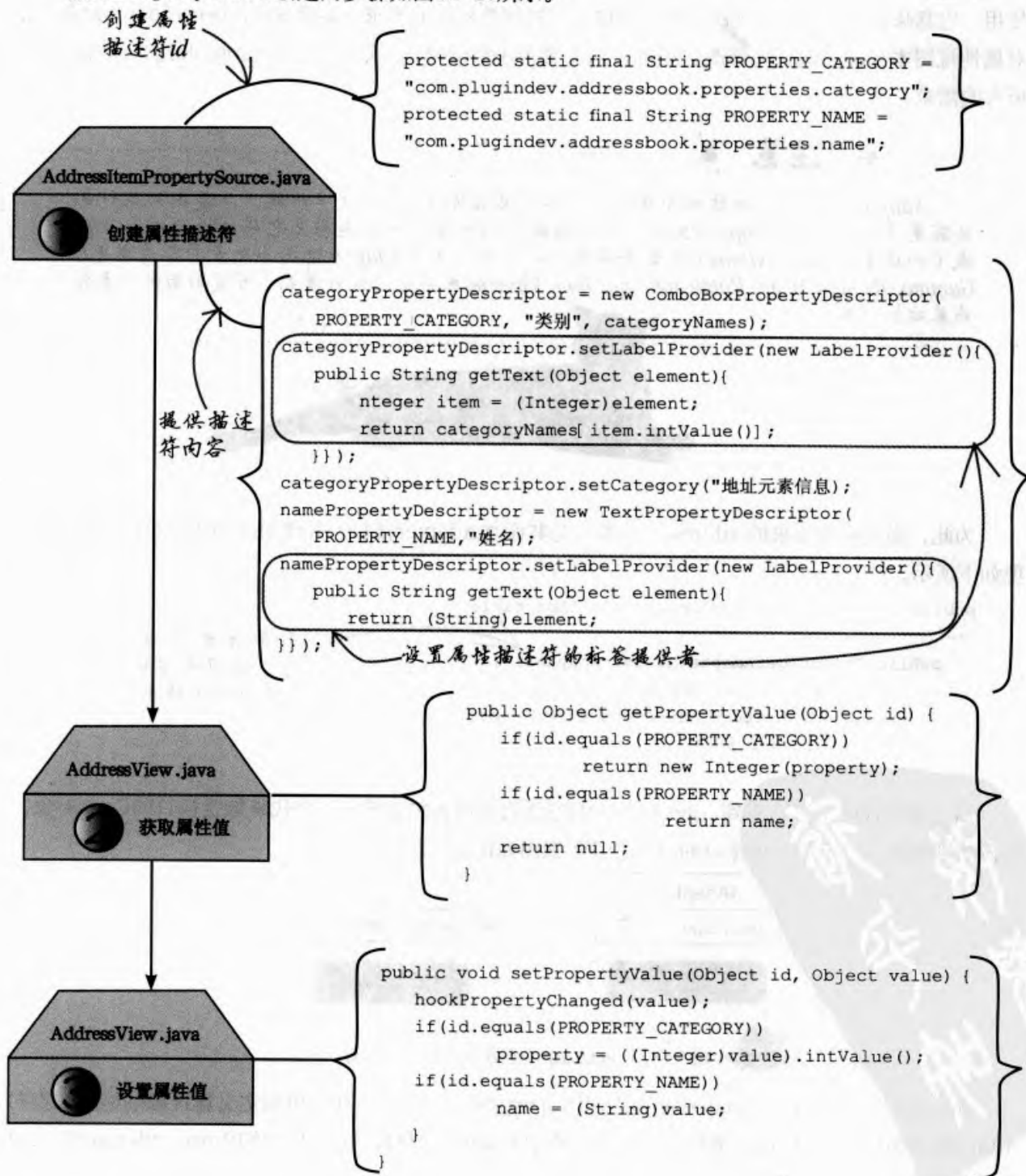
图12-25 getAdapter()方法接口示意图

AddressItemPropertySource是实现了IPropertySource的类，在它里面定义属性源的格式。当向AddressItemPropertySource类传递一个当前模型对象的引用时，这个类允许IPropertySource的实现在需要时获取和设置属性值。

* 注意 *

Eclipse平台提供了对适配器体系结构的支持, 允许开发人员为不同类型的对象定义和注册适配器。插件开发者可以在与需要的适配类型不匹配的情况下, 不返回null, 而是继续请求平台管理器, 由管理器从注册的适配器中寻找相符合的适配类型。

有关IPropertySource创建的步骤如图12-26所示。



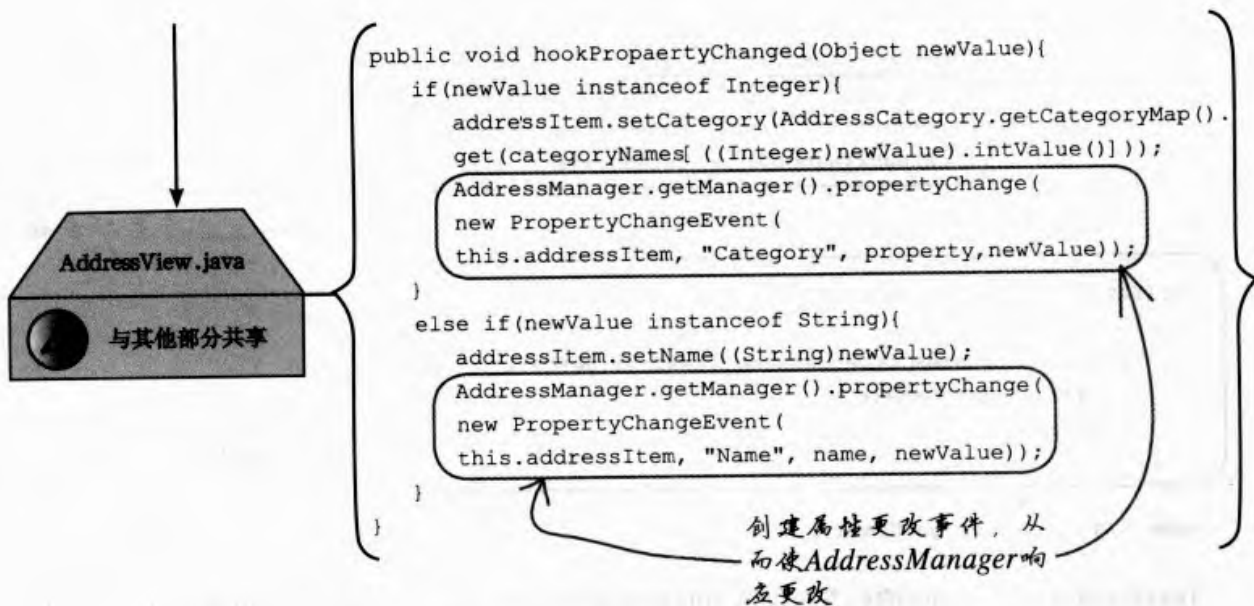


图 12-26 实现IPROPERTYSOURCE的步骤

IPROPERTYSOURCE实现的第一项工作是为每个显示在Properties视图中的项创建一个属性描述符。从图12-6中可以看到，标准的属性包括属性名和属性值。而属性描述符为属性id和属性的显示名称定义String值。AddressItemPropertySource类使用getPropertyDescriptors()方法来请求IPROPERTYSOURCE提供将要显示那些内容的描述符。

开发人员可以实现自己的属性描述符，也可以使用系统提供的属性描述符。系统提供的专用的属性描述符如下所示。

- ★ColorPropertyDescriptor
- ★ComboBoxPropertyDescriptor
- ★TextPropertyDescriptor

这里使用了其中的ComboBoxPropertyDescriptor和TextPropertyDescriptor两个属性描述符来介绍如何提供对属性视图的支持。这两个描述符都支持对属性的编辑。

ComboBoxPropertyDescriptor可以在属性视图的单元格中显示复选框小部件，但必须为其定义标签提供者和一组String数组来为下拉列表填充内容。标签提供者中，方法getText()所使用的参数是方法getPropertyValue()的返回值。在ComboBoxPropertyDescriptor中，这个返回值必须是整数。因为在下拉列表中，每个项是按照它们的索引号来区分的。

因而，在代码中需要在String和整数之间做出某种转换，如下面的代码所示。

```

//类别名称数组，为ComboBoxPropertyDescriptor提供参数
private String[] categoryNames;
//整型变量，代表其在categoryNames数组中的位置
private int property;
//该方法将被构造方法调用来初始化AddressPropertyDescriptor
private void initProperties(){
    if(categoryNames == null)

```



```

{
    categoryNames = new String[ 9 ];
    AddressCategory[] categories = AddressCategory.getTypes();
    for(int i = 0; i < categories.length; i++)
        categoryNames[ i ] = categories[ i ].getCategoryName();
}

for(int i = 0; i < categoryNames.length; i++)
{
    if(categoryNames[ i ].equals(addressItem.getCategory().
        getCategoryName()))
        property = i;
}

name = addressItem.getName();
}

```

将类别名称放入数组

将类别名用整数表示

TextPropertyDescriptor的标签提供器中的参数是String，而名字也是由String表示的，因而不存在这样的问题。

在图12-26的第4步“与其他部分共享”指的是当模型发生改变后，通知其他部分，以便做出相应的更改。为此，需要创建属性改变事件（PropertyChangeEvent），将其传递给AddressManager，并通过AddressManager修改地址本视图中的显示。

由于这个原因，AddressManager也应做出相应更改。为了能接收属性改变事件，它必须实现IPropertyChangeListener接口，代码如下所示。

```

public class AddressManager implements IPropertyChangeListener{
    .....
    public void propertyChange(PropertyChangeEvent event) {
        fireAddressesChanged(AddressItem.NONE, AddressItem.NONE,
            new AddressItem[] { (AddressItem)event.getSource() });
    }
}

```

这样，就完成了对属性视图的支持。添加完属性视图后的效果如图12-27所示。在图12-27中，数字标号表示用户操作的步骤。

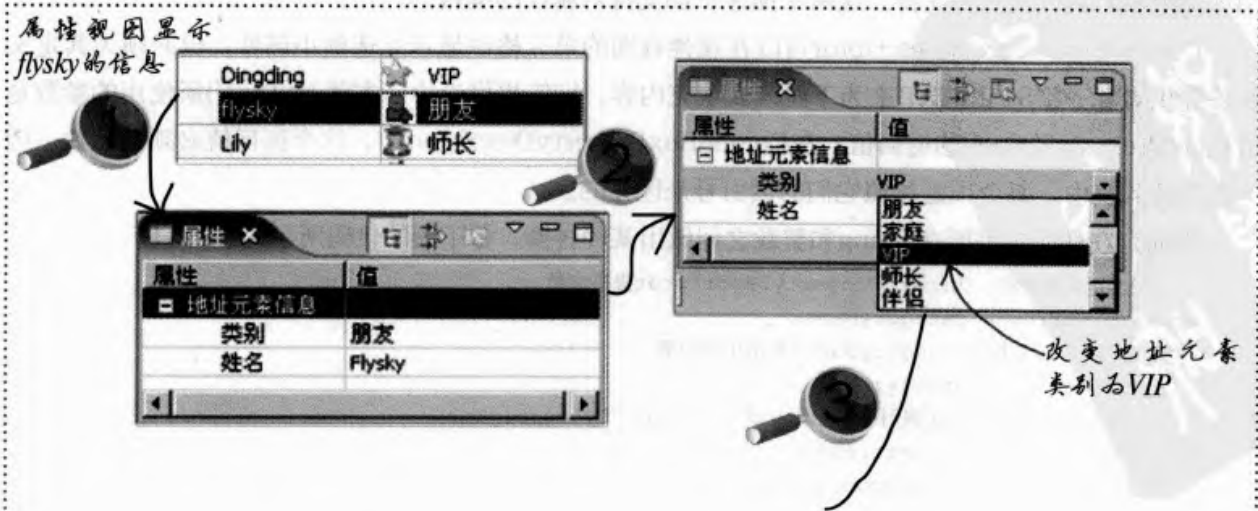
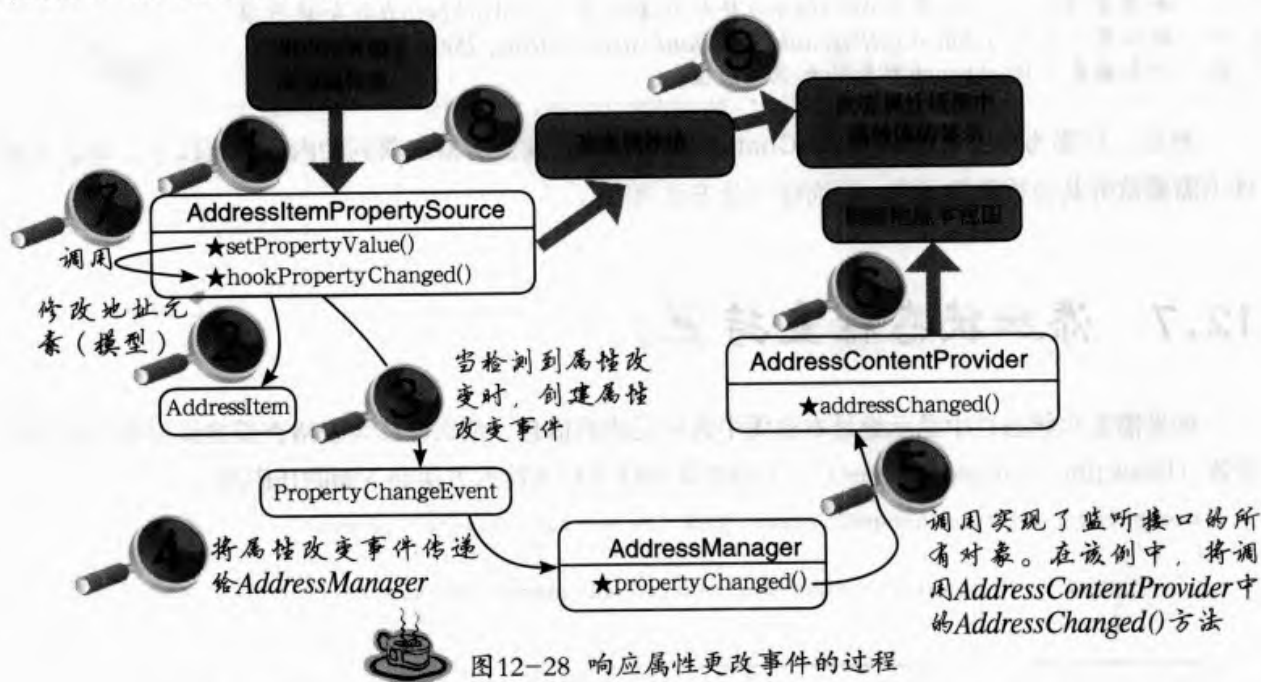




图12-27演示了在界面上如何更改一个地址元素的属性，那么，这些更改是如何发生的呢？接下来的图12-28给出了更多细节。



在图12-28中，数字标号代表动作发生的次序。当用户在属性视图中改变属性值，并且焦点离开该属性所在的表格行后，AddressItemPropertySource中的setPropertyValue()方法被调用，它调用hookPropertyChanged()方法来改变AddressItem的相应属性，并且创建PropertyChangeEvent事件，把该事件传递给AddressManager的propertyChanged()方法，该方法通知所有的监听器（在这个例子中地址本视图的内容提供者是唯一的监听器），使这些监听器中的addressChanged()方法能够得到运行，从而刷新地址本视图，使视图的显示同AddressItem一致。

setPropertySource()还需要改变属性描述符中记录的值，使属性视图的显示同地址本视图的显示一致，二者的显示同模型一致。

12.6.4 监听其他Workbench部分被选中的内容

地址本视图可以监听其他Workbench部分被选中的内容。Eclipse提供的选择监听机制可以使在

任何共享被选中内容的Workbench部分发生的选择事件，都被送到所有可感知的监听器。因此，只要在地址本视图中实现了ISelectionListener接口，就可以监听其他Workbench部分了。因而只需要更改AddressView，让其实现ISelectionListener接口，如下面代码所示。

```
public class AddressView extends ViewPart implements ISelectionListener{
    .....
}
```

实现该接口后，在createPartControl()中，需要进行如下的方法调用来侦听所有Workbench部分被选中的内容，其中的参数为要注册的监听器。

```
//在页面中添加该监听器
getSite().getPage().addSelectionListener(this);
```

※ 注意 ※

如果需要对某个特定的Workbench部分或某种类型的Workbench部分进行监听，那么可以使用getSite().getPage().addSelectionListener(String, ISelectionListener)。第一个参数表示Workbench部分的标识符id值。



然后，只要为视图实现selectionChanged()方法来处理监听器捕获到的内容就可以了。地址本插件不需要监听其他视图的状态，因而保留此方法为空。

12.7 添加状态栏支持

如果需要在状态栏中显示地址本视图中选中元素的信息，那么应该为表格查看器添加选择更改监听器 (ISelectionChangeListener)。为视图添加状态栏支持的方法如下面的代码所示。

```
viewer.addSelectionChangeListener(new ISelectionChangeListener()
{
    public void selectionChanged(SelectionChangedEvent event)
```

```
{
    IStructuredSelection selection =
        (IStructuredSelection) event.getSelection();
```

```
IStatusLineManager statusline = getViewSite().getActionBars()
    .getStatusLineManager();
Object obj = selection.getFirstElement();
if(obj == null)
    return;
if(obj instanceof AddressItem)
{
    AddressItem addressItem = (AddressItem)obj;
```

```
statusline.setMessage(addressItem.getCategory().getImage(),
    addressItem.getName()+" : "
    +addressItem.getCategory().getCategoryName());
```

```
}
```

获得当前站点的
状态栏管理器



设置状态栏消息

```

else
    statusline.setMessage(obj.toString());
}
});

```

状态栏位于Eclipse工作台窗口的底部，地址本视图为状态栏添加支持后的效果如图12-29所示。

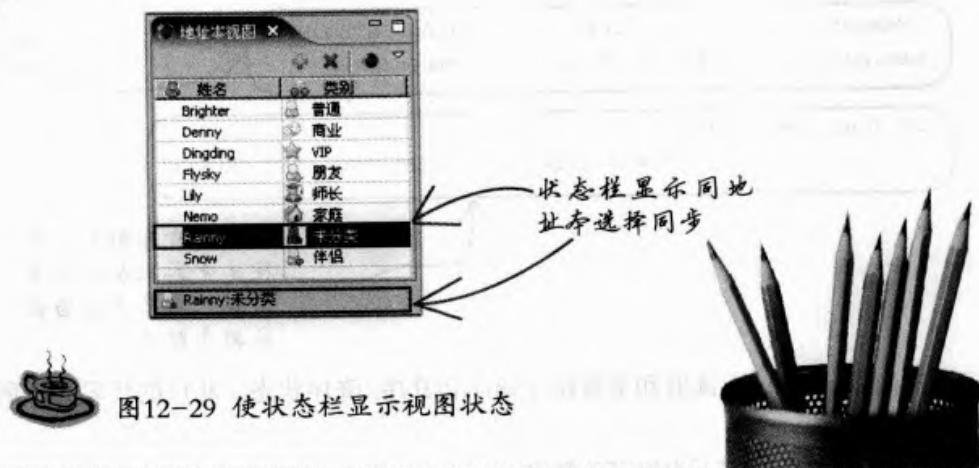


图12-29 使状态栏显示视图状态

12.8 视图状态

到目前为止，当Eclipse会话启动时，地址本视图仅包含当前状态时的项目列表。用户可以使用添加和删除操作，或者编辑属性视图来更改地址本视图中的模型，但只要Eclipse关闭，更改就丢失。另外，应当保存视图的排序和过滤器信息，使得会话重新启动时，可以将视图恢复到同一状态。下面介绍两种不同的实现存储功能的机制。

12.8.1 存储排序和过滤信息

Eclipse使用Memento模式来持久化UI状态，这样，就会在多次会话之间保持用户体验的连贯性。Eclipse的工作台提供了一个通用的Memento接口IMemento，IMemento对象通过创建Memento树来保持对象结构。

保持对象结构用到了Memento机制为过滤和排序提供支持。为了保存排序状态，有必要在FavoritesViewSorter中创建一些标签，这些标签用来作为Memento机制中XML的元素名称，如下所示。

```

private static final String TAG_DESENDING = "descending";
private static final String TAG_INDEX = "columnIndex";
private static final String TAG_CATEGORY = "category";
private static final String TAG_TRUE = "true";

```



添加完标签后，还要为FavoritesViewSorter添加saveState()和init()两个方法。这两个方法分别对应了存储排序信息的两个部分，具体说明如下。

1. 通过把排序顺序和升序/降序状态转换为XML的结构, 将当前的排序状态保存为一个IMemento的实例。相应的代码如下所示。

```
public void saveState(IMemento memento) {
    for (int i = 0; i < infos.length; i++) {
        SortInfo info = infos[i];

        IMemento mem = memento.createChild(TAG_CATEGORY);
        mem.putInteger(TAG_INDEX, info.columnIndex);

        if (info.descending)
            mem.putString(TAG_DESENDING, TAG_TRUE);
    }
}
```

往memento对象中创建一个子元素, 并在其中插入一个整数值

如果是降序排列, 则往其中再添加一个字符串, 用来指示当前的排序规则

2. 从IMemento中读取和重置排序顺序和升序/降序状态。相应的代码如下所示。

```
public void init(IMemento memento) {
    List newInfos = new ArrayList(infos.length);
    IMemento[] mems = memento.getChildren(TAG_CATEGORY);
    for (int i = 0; i < mems.length; i++) {
        IMemento mem = mems[i];
        Integer value = mem.getInteger(TAG_INDEX);
        if (value == null)
            continue;
        int index = value.intValue();
        if (index < 0 || index >= infos.length)
            continue;
        SortInfo info = infos[index];
        if (newInfos.contains(info))
            continue;
        info.descending =
            TAG_TRUE.equals(mem.getString(TAG_DESENDING));
        newInfos.add(info);
    }
    for (int i = 0; i < infos.length; i++)
        if (!newInfos.contains(infos[i]))
            newInfos.add(infos[i]);
    infos = (SortInfo[]) newInfos.toArray(
        new SortInfo[ newInfos.size() ] );
}
```



在应用程序关闭时, FavoritesViewSorter中的saveState()方法运行, 由这个方法将所要保存的信息填入IMemento对象之中。在下一应用程序打开时, FavoritesViewSorter使用视图的init()方法接收工作台传递的IMemento对象。init()方法和saveState()方法是由抽象类ViewPart定义的。

完成了init()和saveState()后, 还要在AddressView中为IMemento设置一个字段, 并且继承

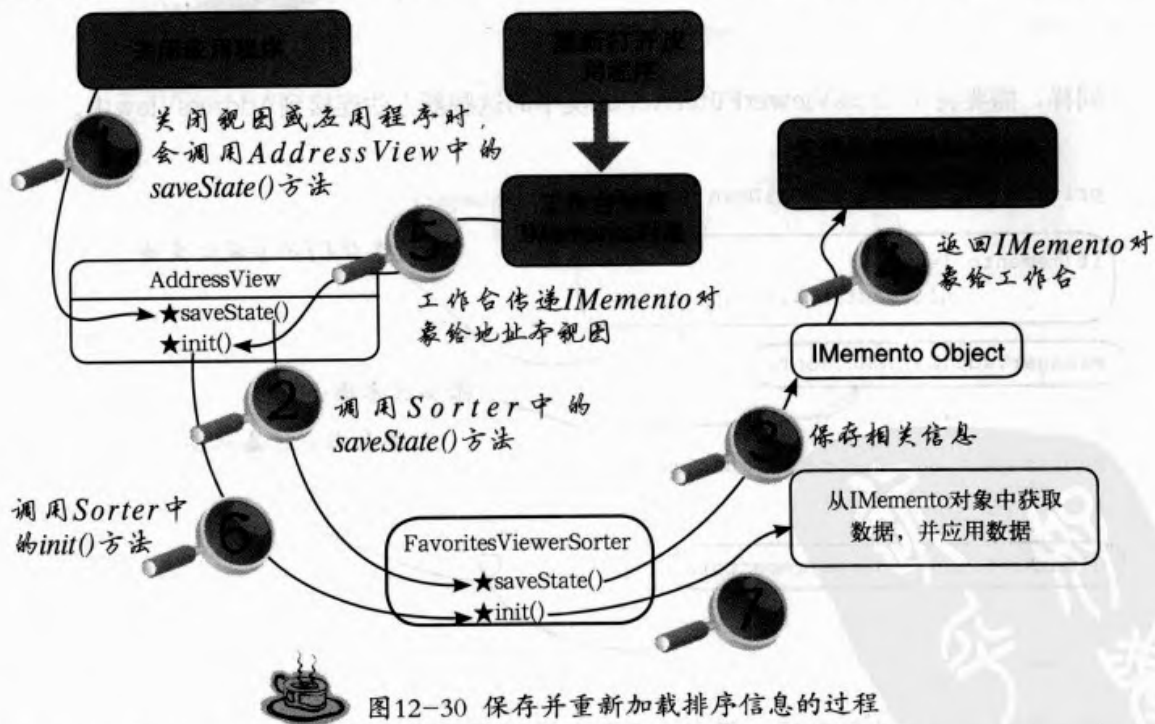
ViewPart中的init()方法和saveState()方法来获取和保存IMemento对象，代码如下所示。

```
private IMemento memento;
public void init(IViewSite site, IMemento memento) throws PartInitException
{
    super.init(site, memento);
    this.memento = memento;
}
public void saveState(IMemento memento){
    super.saveState(memento);
    sorter.saveState(memento);
}
```

然后，在AddressView中的createTableSorter()中添加如下代码。

```
//设置排序器
private void createTableSorter() {
    .....
    if(memento != null)
        sorter.init(memento);
    viewer.setSorter(sorter);
}
```

保存并重新加载排序信息的过程如图12-30所示，图中的数字标号代表动作发生的先后顺序。



同样，要实现对过滤器状态的保存，也需要在AddressViewerCategoryFilter类中加入如下两个方法。

```
private static final String TAG_PATTERN = "pattern";
private static final String TAG_TYPE = "CategoryFilterInfo";
```



```

.....
public void saveState(IMemento memento) {
    if (pattern.length() == 0)
        return;
    IMemento mem = memento.createChild(TAG_TYPE);
    mem.putString(TAG_PATTERN, pattern);
}

public void init(IMemento memento) {
    IMemento mem = memento.getChild(TAG_TYPE);
    if (mem == null)
        return;
    setPattern(mem.getString(TAG_PATTERN));
}

```



由于过滤是通过用户选择过滤操作后开始运行的，因而需要在AddressViewerFilterAction类中加入如下方法来调用AddressViewerCategoryFilter类中的新方法。

```

public void saveState(IMemento memento){
    categoryFilter.saveState(memento);
}

public void init(IMemento memento){
    categoryFilter.init(memento);
}

```



同样，需要将AddressViewerFilterAction类中的这些新方法连接到AddressView中。

```

private void fillLocalPullDown(IMenuManager manager) {

```

```

    if (memento != null)
        filterAction.init(memento);

```

```

    manager.add(filterAction);

```

```

}

public void saveState(IMemento memento){

```

```

    .....

    filterAction.saveState(memento);
}

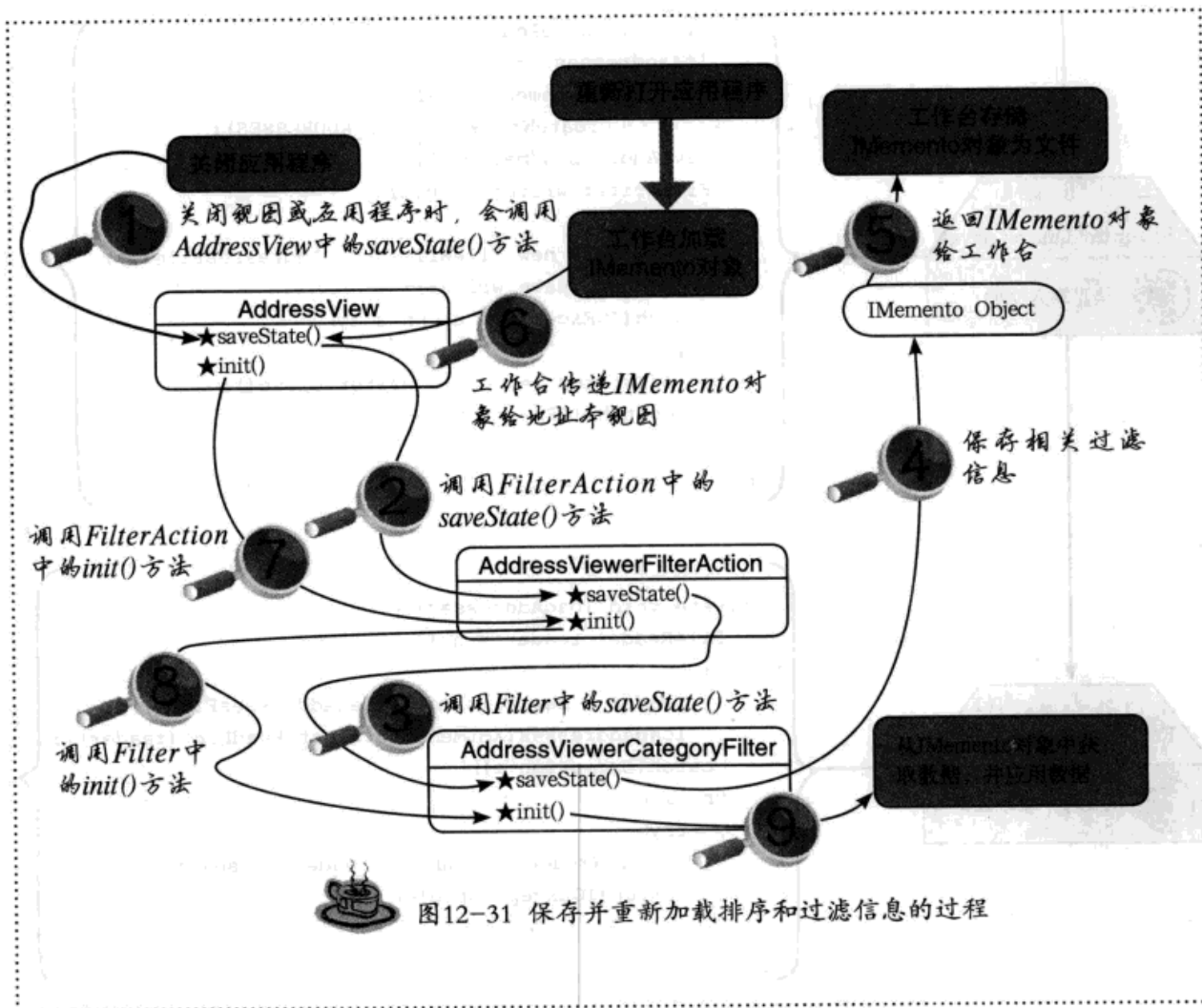
```

为持久化UI状态提供支持

添加过滤操作

保存当前过滤状态

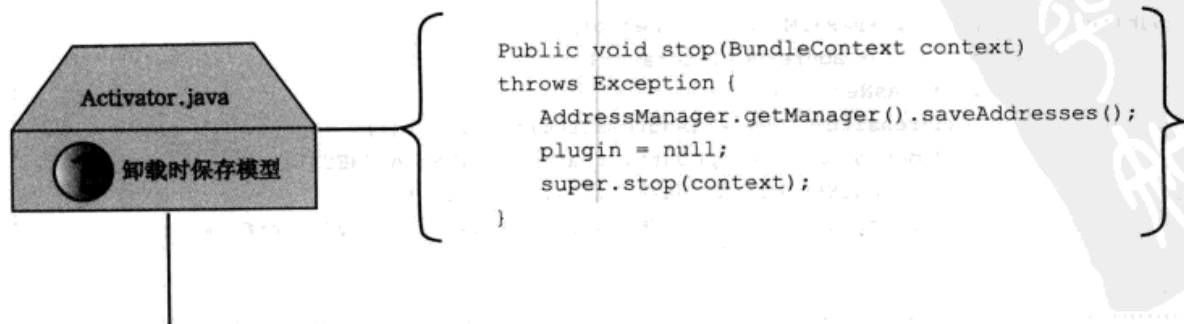
增加了过滤状态的保存后，保存并重新加载视图信息的过程如图12-31所示。在此图中，相同步骤的动作代表同一类型的操作。同保存排序状态相比，保存过滤状态需要通过操作AddressViewerFilterAction调用AddressViewerCategoryFilter中的方法来完成，因为AddressView中并没有直接使用到过滤器对象（视图是同排序器是一一对应的，但却可以有多个过滤器）。AddressViewerFilterAction只简单调用AddressViewerCategoryFilter，因而没有用数字标号标出。



12.8.2 存储视图元素信息

地址元素存储的信息应该同视图的排序和过滤状态分离开来。视图的排序和过滤的存储信息可以通过工作台传递，但是地址元素如果同它们放在一起结构就比较混乱，而且排序和过滤是显示信息，地址元素的信息是模型信息，应该分别存储。因而，本节为地址元素创建新的Memento，并把它存储到项目工作空间中的XML文件里。

为此，在地址本插件中增加如图12-32所示步骤。



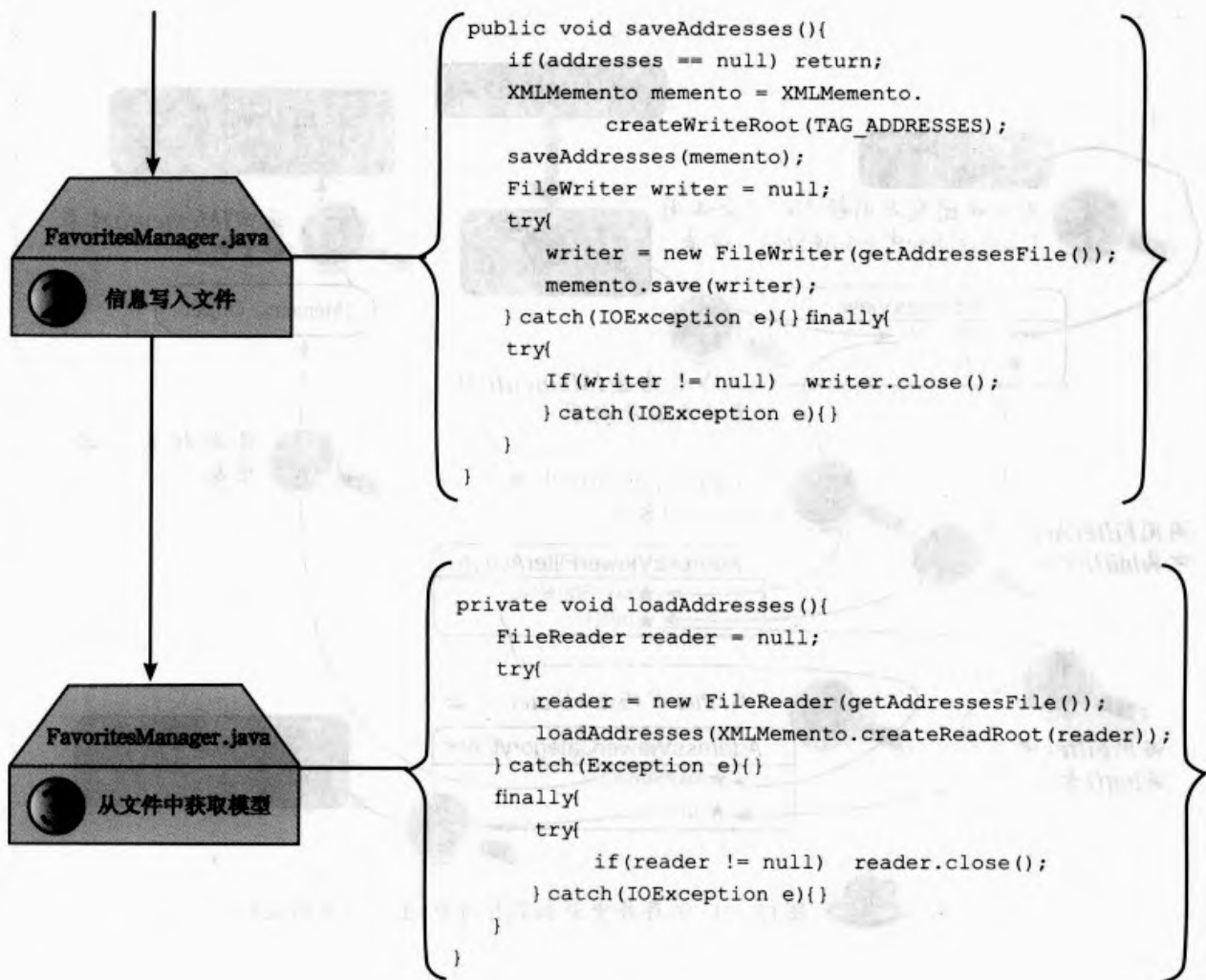


图12-32 存储模型元素信息的步骤

在图12-32中的第●步里，Activator的stop()方法中加入了保存模型的代码，保证了插件在卸载时保存模型信息。

第●步“信息写入文件”中，首先将地址元素信息存储在XMLMemento对象中，然后再将该对象保存到指定的文件里。下面的代码是将地址元素信息存储在XMLMemento对象中的方法。

```
public void saveAddresses(IMemento memento){
    Iterator iter = addresses.iterator();
    while(iter.hasNext()){
        AddressItem item = (AddressItem)iter.next();
        IMemento child = memento.createChild(TAG_ADDRESS);
        child.putString(TAG_NAME, item.getName());
        child.putString(TAG_CATEGORY, item.getCategory().getCategoryName());
    }
}
```

指定的文件名称为Addresses.xml，在工作空间元数据的如下子目录中可以找到它。

```
<workspaceDir>\.metadata\plugins\com.plugindev.addressbook
```

获得该文件的代码如下所示。

```
private File getAddressesFile() {  
    return Activator.getDefault().getStateLocation().  
        append("Addresses.xml").toFile();  
}
```

同样，第 ❶ 步“从文件获取模型”也需要首先由Addresses.xml文件的信息构造XMLMemento对象，然后利用该对象构造地址元素，下面的代码是根据XMLMemento对象构造地址元素的方法。

```
private void loadAddresses(XMLMemento memento)  
{  
    IMemento[] children = memento.getChildren(TAG_ADDRESS);  
    for(int i = 0; i < children.length; i++)  
    {  
        AddressItem item =  
            createNewAddressItem(children[i].getString(TAG_NAME),  
                                children[i].getString(TAG_CATEGORY));  
        if(item != null)  
            addresses.add(item);  
    }  
}
```

Addresses.xml文件的内容很简单，XML元素名称便是构造的标记，如下面代码所示。

```
<?xml version="1.0" encoding="UTF-8"?>  
<Addresses>  
    <AddressItem category="师长" name="Lily"/>  
    <AddressItem category="VIP" name="publish"/>  
    <AddressItem category="VIP" name="Dingding"/>  
    <AddressItem category="未分类" name="Rainny"/>  
    <AddressItem category="商业" name="Denny"/>  
    <AddressItem category="家庭" name="Nemo"/>  
    <AddressItem category="普通" name="Brighter"/>  
</Addresses>
```

12.9 加载和卸载图标

在创建视图的过程中，地址本插件加载了许多图标，这些图标在插件代码中以Image对象的形式被使用。Image对象是一个包装了本地资源的Java结构，它不能被Java的垃圾回收器管理。因而，如果在加载了图像之后不卸载，可能会导致内存泄漏。



Eclipse使用ImageDescriptor来描述Image, 这是一个纯Java类型, 因而不需要被显式地管理和删除。ImageDescriptor可以创建图像, 在插件或其他应用程序中为了避免到处创建图像, 最好能够有一个专门管理图像的对象, 由其创建图像的唯一实例。为此, 地址本插件中创建了ImageCache类来管理和删除Image图像。代码如下所示。

```
package com.plugindev.addressbook.util;
import java.util.HashMap;
import java.util.Iterator;
import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.swt.graphics.Image;
public class ImageCache {
    private final HashMap<ImageDescriptor, Image> imageMap
= new HashMap<ImageDescriptor, Image>();
    //ImageCache实例

private static ImageCache instance;
    public static ImageCache getInstance(){
        if(instance == null)
            instance = new ImageCache();
        return instance;
    }

    public Image getImage(ImageDescriptor descriptor)
    {
        if(descriptor == null)
            return null;

        Image image = (Image)imageMap.get(descriptor);
        if(image == null)
        {
            image = descriptor.createImage();
            imageMap.put(descriptor, image);
        }

        return image;
    }

    public void dispose(){
        Iterator iter = imageMap.values().iterator();
        while(iter.hasNext())
            ((Image)iter.next()).dispose();
        imageMap.clear();
    }
}
```

使用getInstance()方法
获得ImageCache的
唯一实例

保证在插件中每个图像
资源只有唯一的创建图
像操作

ImageCache()类使用了同AddressManager相同的模式(单例模式)来获得ImageCache的唯一全局实例, 这就保证了在整个插件中只有这一个实例来创建和删除图像, 从而有效地避免了潜在的内存泄漏问题。

dispose()方法需要在插件卸载时执行。因此, 还需要在插件类的卸载方法中加入如下代码。

```
public void stop(BundleContext context) throws Exception {
```

```
.....  
ImageCache.getInstance().dispose();  
.....  
}
```

获得这些图像的描述符可以使用插件类中的getImageDescriptor()方法,该方法以图像资源的相对路径作为参数,因而,地址本插件创建了ImageKey来保存所有图像路径的描述字符串。

12.10 本章小结

本章介绍了创建一个视图的过程,包括如何创建新视图,如何响应其他视图中的选择来修改视图,以及如何保存视图中的状态信息等。视图是Eclipse工作台的基本用户界面组件,从本章可以看到,创建视图非常容易,但是必须要遵循Eclipse插件体系结构规定的方法。

视图的扩展也非常方便。本章介绍了如何使用编程方式为Eclipse的视图创建新的操作,这些新的操作可以是在提供视图的插件中定义,也可以在其他插件中定义。因而可以轻松地针对不同使用者的需求增加功能,完善视图。

视图的真正价值在于同用户通信,用户通过与视图的交互将变更反映到视图所代表的模型中,完成某项功能。而且,从本章例子中可以看到,视图可以同其他工作台部分相互通信,通过通信,可以为工作台的其他部分(如本章中实现的属性视图)提供定制的内容,并且反应到视图的显示上来。

下一章将讨论如何使用编辑器,它是另外一种类型的工作台部分。



我们把谈论的结果
记录下来吧!

第13章 编辑器 (Editors)

拉开崭新的学习帷幕

上一章介绍了“视图”并为地址本插件添加了一个功能完整的视图部分，接下来要学习Eclipse插件开发中最复杂也是最主要的部分——编辑器。本章将创建一个新的“地址本”编辑器，这个地址本编辑器模仿Eclipse PDE，采用富客户端表现方式，利用各种界面元素来展示要添加的内容。

本章内容包括：

- ★Eclipse编辑器体系结构概览和Eclipse中的编辑器。
- ★声明和创建一个新的Form编辑器，并实现多个页面。
- ★为编辑器创建数据模型，并为模型提供响应编辑器更改的方式。
- ★介绍并实现编辑器的生命周期。
- ★为编辑器添加操作，介绍管理编辑器操作的方法。



进入第13章

13.1 Eclipse编辑器体系结构概览

编辑器是用户创建和修改资源最主要的机制，在这些资源里存储了用户需要的信息，因而，编辑器可以说是应用程序的核心部分，Eclipse用户界面框架的设计也是以编辑器为中心的，这点在下一章“透视图”的介绍中读者将会有更深的印象。Eclipse提供了一些基本的编辑器，如文本和Java源代码编辑器，还提供了一些比较复杂的多页编辑器，PDE插件清单编辑器就是其中的一个典型范例。当插件需要自己定义格式并且使编辑器支持自定义格式时，可以使用Eclipse提供的编辑器扩展点来扩展现有的Eclipse编辑器。

在此之前，有必要先浏览Eclipse编辑器的继承体系，简单了解Eclipse提供的各种编辑器的使用和扩展方式。

编辑器必须实现org.eclipse.ui.IEditorPart接口，通常编辑器都作为该接口的默认扩展org.eclipse.ui.part.EditorPart的子类，因而它也是org.eclipse.ui.part.WorkbenchPart的间接子类，它继承了用来实现IEditorPart接口的很多行为，如图13-1所示。

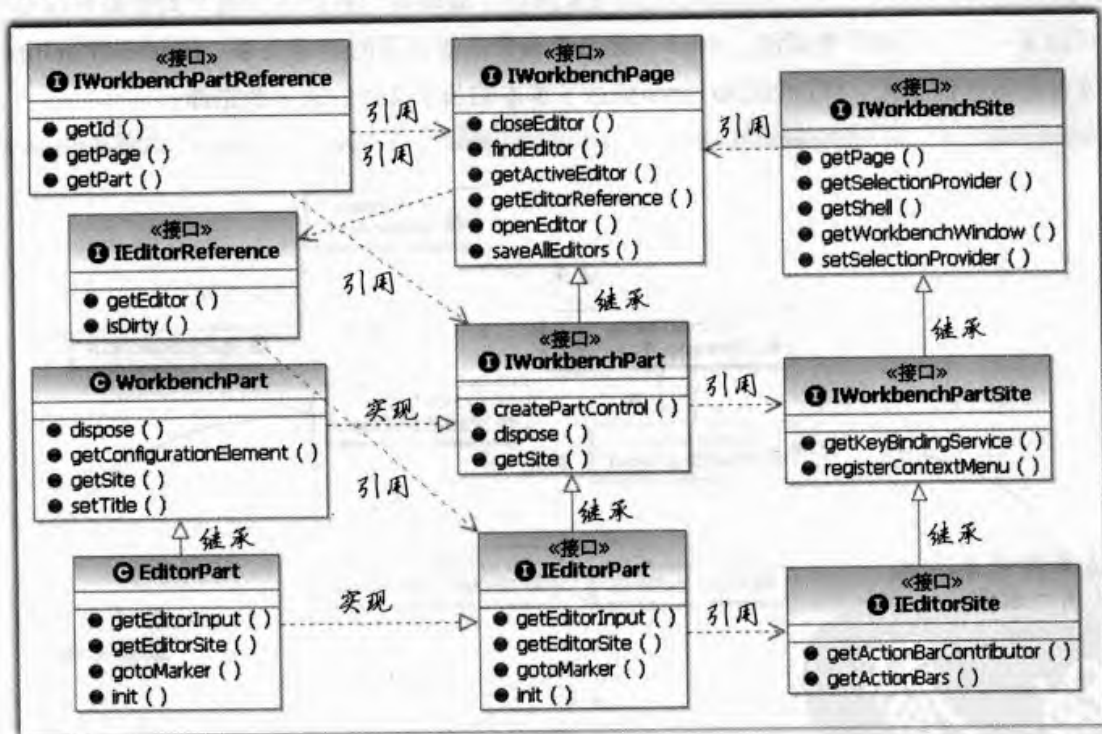


图13-1 EditPart类图以及调用关系

编辑器必须包含在org.eclipse.ui.IEditorSite中，org.eclipse.ui.IEditorSite又包含在工作台页面中。同视图一样，在工作台页面中保存了编辑器的参考信息（org.eclipse.ui.IEditorReference）的实例，通过IEditorReference所提供的方法getEditor()方法来间接获取编辑器。这样避免了工作台页面直接保存编辑器本身，从而使引用编辑器同加载定义编辑器的插件相分离，可以避免或延迟相关插件的加载。

编辑器同视图通过超类org.eclipse.ui.part.WorkbenchPart和org.eclipse.ui.part.

IWorkbenchPart接口共享一个共同的行为集合，但是，它们也有非常重要的区别（参见第12.1节）。在本章中，通过构建地址本编辑器，读者将会进一步认识这些区别。

13.2 Eclipse工作环境中的编辑器

Eclipse工作环境中预定义了一些编辑器，这些编辑器通常都是基于文本（Text）的，还有一些编辑器是基于图形的，如使用Eclipse的GEF（Graphical Editing Framework，图形编辑框架）技术定制的编辑器和使用Eclipse的GMF（Graphical Modeling Framework，图形建模框架）生成的编辑器等。本章主要讨论基于文本的编辑器，有关基于图形编辑器的相关内容请参阅

<http://www-128.ibm.com/developerworks/cn/opensource/top-projects/eclipse.html>中的GEF和GMF的相关栏目。

从图13-1中可以知道，所有工作空间中的编辑器都是派生自抽象类EditorPart，这个类是接口IEditorPart的默认实现类。接口IEditorInput抽象描述了编辑器的数据源。这个数据源可以是一个文件，也可以是一个二进制的流。图13-2显示了编辑器输入源的继承关系。IPathEditorInput描述了本地文件的数据源格式；IFileEditorInput描述了基本的基于文件的输入数据源。



图13-2 编辑器输入源的继承关系

当编辑器被创建之后，这些编辑器输入源通过调用编辑器中的init()方法来设定。它可以由getEditorInput()的方法重新获得。

图13-3显示了基于文本的编辑器的继承关系。EditorPart编辑器的子类通常重写其中的createPartControl()方法，使用这个方法可以创建必要的SWT和JFace组件来构成特定的编辑器界面。

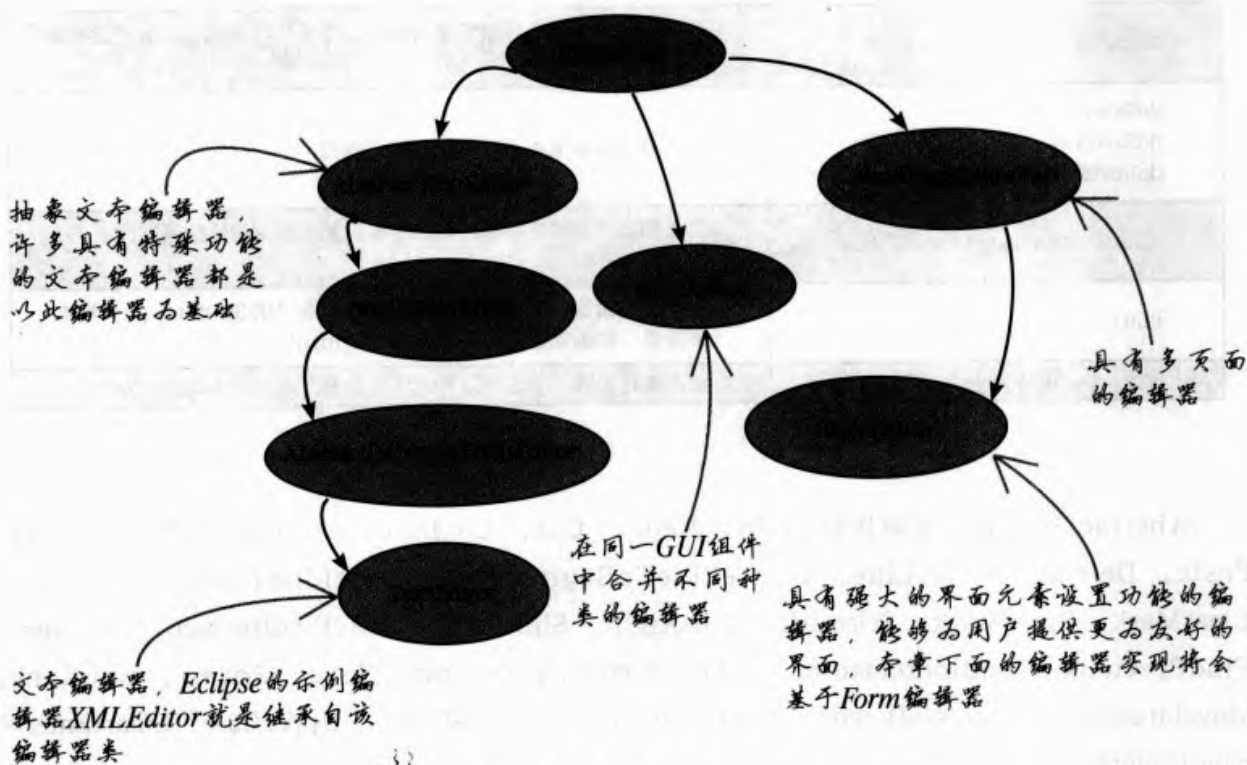


图13-3 Eclipse提供的文本编辑器体系

下面简单介绍几个重要的编辑器类。

13.2.1 AbstractTextEditor类

AbstractTextEditor类为基于文本的编辑器实现了许多基本功能，如下面所示几种。

- ★文本处理的标准函数，如剪切、拷贝、粘贴、查找和替换等。
- ★上下文菜单的管理。
- ★响应工作空间中的资源更改，如工程关闭、资源删除等。

AbstractTextEditor将文本模型和显示分离开。当前文本为编辑器提供IDocumentProvider的实例，这样就允许不同编辑器访问同一个文本模型。IDocumentProvider实例还负责保存正在使用的文本。当在AbstractTextEditor中执行doSave()和doRevertToSaved()等保存操作时，AbstractTextEditor将调用IDocumentProvider实例的相应方法。

AbstractTextEditor的子类可以重写一些方法来扩展AbstractTextEditor，从而调整其行为，通常需要重写的方法如表13-1所示。

表13-1 扩展AbstractTextEditor需要重写的方法

方法名称	方法描述
createActions()	创建AbstractEditor的标准操作，子类可以扩展该方法来添加自定义操作
createPartControl()	创建界面相关显示。子编辑器类需要重写此方法来定义特定的界面显示
dispose()	当子编辑器类需要释放资源（颜色、字体、打印机等）时，需扩展此方法
doSave() doSaveAs() doRevertToSave()	这些方法保存文本模型并且存储该模型
editorContextMenuAboutToShow()	当在编辑区域单击右键时，将会执行此方法来构建编辑器的上下文菜单。
Init()	初始化编辑器。为编辑器提供编辑器站点（IEditorSite，由扩展点来获得）和编辑器输入实例（IEditorInput）
isSaveAsAllowed()	标准实现为假，子类可以更改实现为真以启用doSaveAs()方法

Abstract实现的标准操作有Undo, Redo, Cut, CutLine, CutLineToEnd, Copy, Paste, Delete, DeleteLine, DeleteLineToBeginning, DeleteLineToEnd, SetMark, ClearMark, SwapMark, SelectAll, ShiftRight, ShiftLeft, Print, FindReplace, FindNext, FindPrevious, FindIncremental, FindIncrementalReverse, Save, Revert, GotoLine, MoveLinesUp, MoveLinesDown, CopyLinesUp, CopyLinesDown, UpperCase, SmartEnter和SmartEnterReverse。

13.2.2 MultiEditor类

MultiEditor类在一个GUI组件中合并了不同的编辑器，当开发者需要使用不同类型的编辑器显示同一个资源模型时，可以使用这个类。如果需要这样做，则管理不同的编辑器（MultiEditor将其称作内部编辑器）可能需要使用如表13-2所示的方法。

表13-2 MultiEditor中的方法

方法名称	方法描述
createInnerPartControl()	创建一个内部编辑器的GUI
getActiveEditor()	返回当前正在使用的编辑器
getInnerEditors()	返回所有的内部编辑器



13.2.3 MultiPageEditorPart 类

抽象类MultiPageEditorPart用来实现多页面编辑器。每个页面包含它自己的编辑器，就像PDE编辑器那样。MultiPageEditorPart的子类必须实现如表13-3所示的方法。

表13-3 MultiPageEditorPart中的方法

方法名称	方法描述
createPages()	此方法创建所有的编辑器页面，也可以使用addPage()方法来实现此功能
IEditorPart.doSave() IEditorPart.doSaveAs()	这两个方法保存编辑器的所有内容
IEditorPart.isSaveAsAllowed()	如果需要使doSaveAs()方法可用，此方法应该返回真值

13.2.4 FormEditor 类

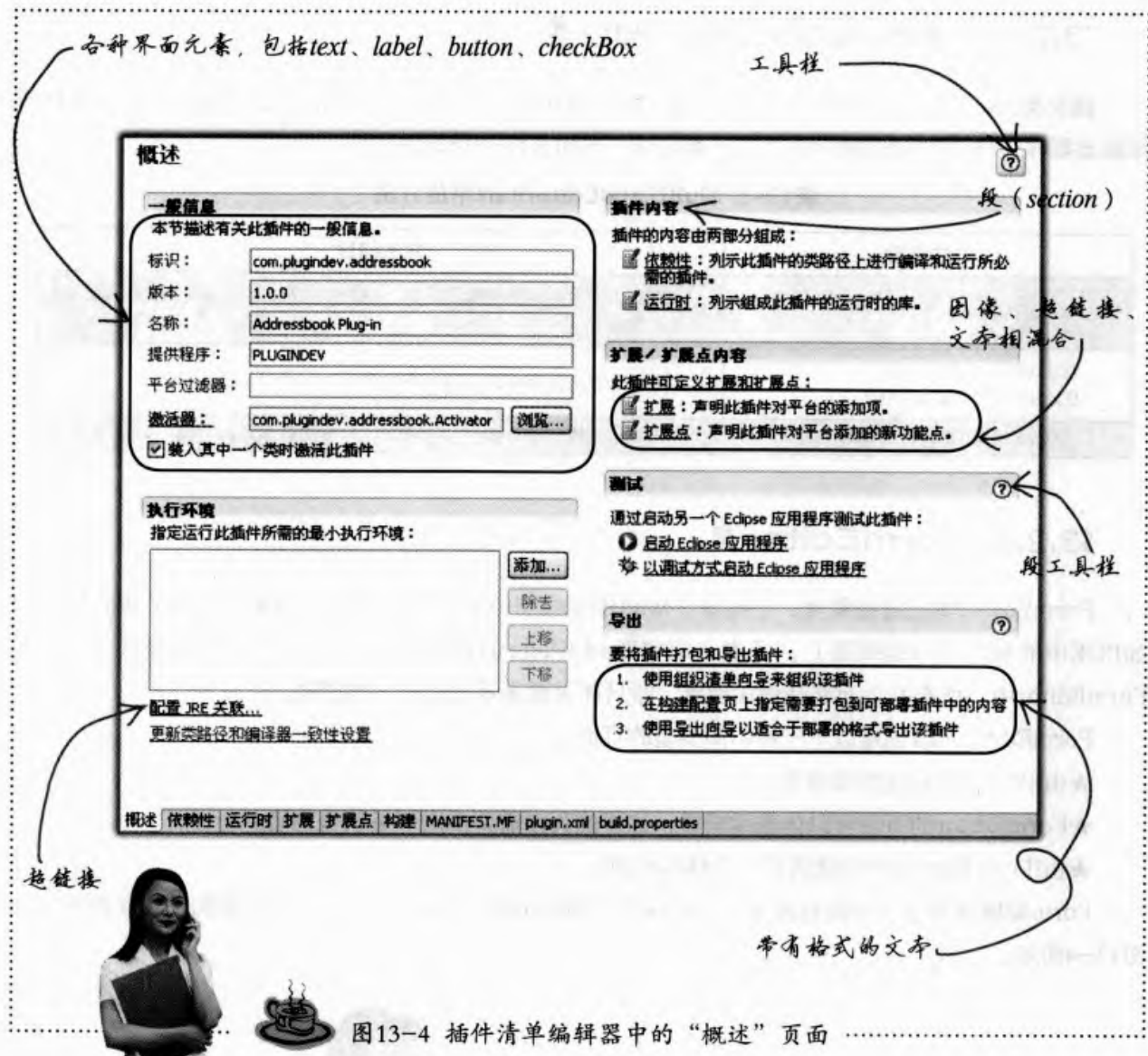
FormEditor类也是抽象类，它继承了MultiPageEditorPart类，用来实现基于表格的编辑器（例如PDE中的插件清单编辑器）。子类必须实现addPages()方法来为FormEditor填充编辑页面。在FormEditor中，所有的页面都是懒式创建，即只在需要显示它们时才被创建。

FormEditor可以创建以下三种不同类型的页面。

- ★由SWT元素构建的普通页面。
- ★FormEditor的页面可以内嵌编辑器，这样的页面需要由内部编辑器构建。
- ★由IFormPage实例构建的基于表格的页面。

Form编辑器的显示功能非常强大，在这里以插件清单编辑器（由Form编辑器构建）来说明。如图13-4所示。





在图13-4中, 插件清单的“概述”页面容纳了SWT组件、超链接和图片, 并且当尺寸不够时能够滚动 (就像Web浏览器展示的效果一样)。为了看起来和文本内容更加融合, 所有组件都是平滑的 (flat)。

由于Form编辑器具有丰富的显示效果, 可以模仿HTML的显示, 并同时容纳不同的组件、超级链接、图片、文本, 本章将为地址本插件创建一个Form编辑器, 用来填充每个地址元素的详细信息。在地址本编辑器中, 实现的功能比较简单, 但是对于读者掌握编辑器的基本设计思路以及Form编辑器的使用来说, 已经足够了。效果如图13-5所示。

※ 注意: ※

Eclipse Form技术不仅可以用于编辑器, 也可以用于视图、对话框、向导等任何UI形式。Form的概念和编辑器、视图等是一个层次的概念。



13.3 为例子增加一个编辑器

从本节开始，本书将介绍如何创建一个多页表格编辑器。

同创建视图的步骤类似，创建一个编辑器分为以下两步。

✚ 在插件清单中定义编辑器；

✚ 创建包含代码的编辑器部件。

在创建插件的同时使用模板创建编辑器，就可以一次性地完成所有步骤。这与在第10章中创建视图的步骤类似（参见第10.1节“创建插件工程”）。如果插件已经存在了，那么创建编辑器就需要完成这两个步骤。

13.3.1 声明编辑器

创建编辑器的第一步是在清单中定义编辑器。在插件清单编辑器的“扩展”页上，单击“添加”按钮，选择org.eclipse.ui.editors，然后单击“完成”按钮，编辑器扩展就添加到了“所有扩展”段区域中。

接着，右键单击org.eclipse.ui.editors扩展，并选择“新建”→“编辑器”命令，添加一个编辑器扩展。双击此新编辑器扩展，打开属性视图，在其中输入如图13-5所示的值，地址本表格编辑器的声明就完成了。

由图13-5可以看到，编辑器扩展有如下所示的属性。

★id*——编辑器的唯一标识符。

★name*——编辑器的易读名称。

★icon——编辑器左上角显示的图像。在该属性区域填写相对于插件安装目录的路径。

★extensions——文件扩展名组成的一个字符串，中间由逗号分隔，指示编辑器可接受的文件类型。在地址本插件编辑器例子中，将该区域置空。

★class——定义编辑器和实现org.eclipse.ui.IEditorPart的类的完全限定名。

★command——启动外部编辑器的命令。可执行命令必须位于系统路径或插件的目录中。

★launcher——实现org.eclipse.ui.IEditorLauncher并打开外部编辑器的类的名字。

★contributorClass——实现org.eclipse.ui.IEditorActionBarContributor的类的完全限定名。这个类反映了编辑器类型的特征。只有在定义了class属性元素时，才能定义该属性。

★default——取值为true或false。如果为true，编辑器将作为该类型的默认编辑器。只有在为同一类型注册了多个编辑器时，该属性才有意义。如果编辑器不是该类型的编辑器，仍然可以使用“打开方式”子菜单，为选择的资源启动该编辑器。

★filenames——由逗号分隔的文件名组成的字符串，指示编辑器所理解的文件名。例如，理解插件和片断清单文件的编辑器可以注册plugin.xml,fragment.xml。

★symbolicFontName——定义字体的符号名称。这个名称需要在org.eclipse.ui.fontDefinitions中定义。如果没有定义或定义无效，则字体名字是编辑器外观中org.eclipse.jface.textfont的存储值。

★matchingStrategy——实现org.eclipse.ui.IEditorMatchingStrategy类的完全限定名。只有在

定义了class属性元素时,才能定义该属性。这个属性允许编辑器提供自己的匹配算法来匹配特定的编辑器输入。

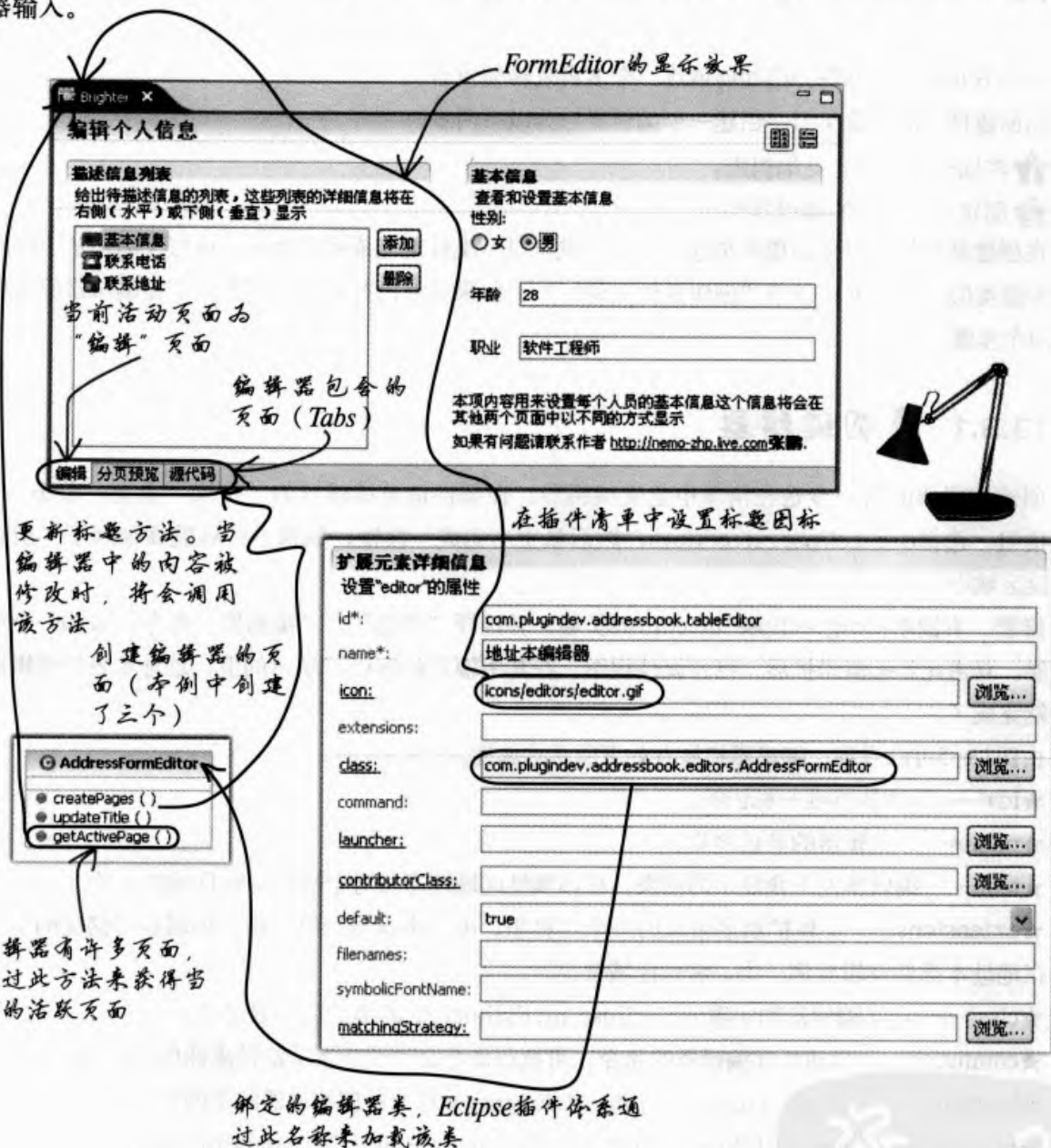


图13-5 插件清单中的编辑器声明

13.3.2 创建编辑器

在本章的引言部分曾经提到, 编辑器行为由实现org.eclipse.ui.IEditorPart接口的类定义, 通常通过创建抽象类org.eclipse.ui.part.EditorPart或org.eclipse.ui.part.MultiPageEditorPart类的子类来实现该接口。

地址本表格编辑器将扩展MultiPageEditorPart的子类FormEditor, 并为用户提供三个页面来编

辑及显示它的内容。

FormEditor提供了抽象方法addPages(), 必须要实现该方法。另外, 为了保存特定的编辑器信息, 还需要重写以下三个方法。

★doSave()

★doSaveAs()

★isSaveAsAllowed()

地址本插件的编辑器不需要实现“另存为...”的功能, 因此将doSaveAs()方法留空, 为isSaveAsAllowed()方法返回false。后续部分将介绍doSave()方法的实现, 因而, 在此也将该方法留空。创建编辑器的代码如下所示。

```
public class AddressFormEditor extends FormEditor {
    protected void addPages() {
        try {
```

```
        addPage(new MasterDetailsPage(this));
        addPage(new PageWithSubPages(this));
        addPage(new SourcePage(this));
```

为编辑器添加三个编辑器页面。这三个页面将在第13.5节中分别介绍

```
    } catch (PartInitException e) {
        e.printStackTrace();
    }
```

```
    updateTitle();
}
```

```
void updateTitle(){
    IEditorInput input = getEditorInput();
    setPartName(input.getName());
    setTitleToolTip(input.getToolTipText());
}
```

更新编辑器标题。其中包括标题名称, 标题浮动提示名称等。getEditorInput()为继承自EditorPart类的方法。此时EditorInput还未定义, 本书将在13.2.3节介绍

```
@Override
public void doSave(IProgressMonitor monitor) {
}

@Override
public void doSaveAs() {
}

@Override
public boolean isSaveAsAllowed() {
    return false;
}
```

13.3.3 创建编辑器输入

编辑器输入的创建非常简单, 只要创建一个继承FormEditorInput()的类SimpleFormEditorInput即可。在FormEditorInput类中有一些默认的方法, 在地址本编辑器的例子中, 不需要修改这些方法,

只要简单继承即可。另外，为了管理编辑器所使用的模型（信息存储在该模型中），本书还将在第13.4.2节“数据管理模型”一节创建AddressListManager类，并在SimpleFormEditorInput中引用该AddressListManager类的对象作为编辑器的输入。FormEditorInput和SimpleFormEditorInput的关系如图13-6所示。

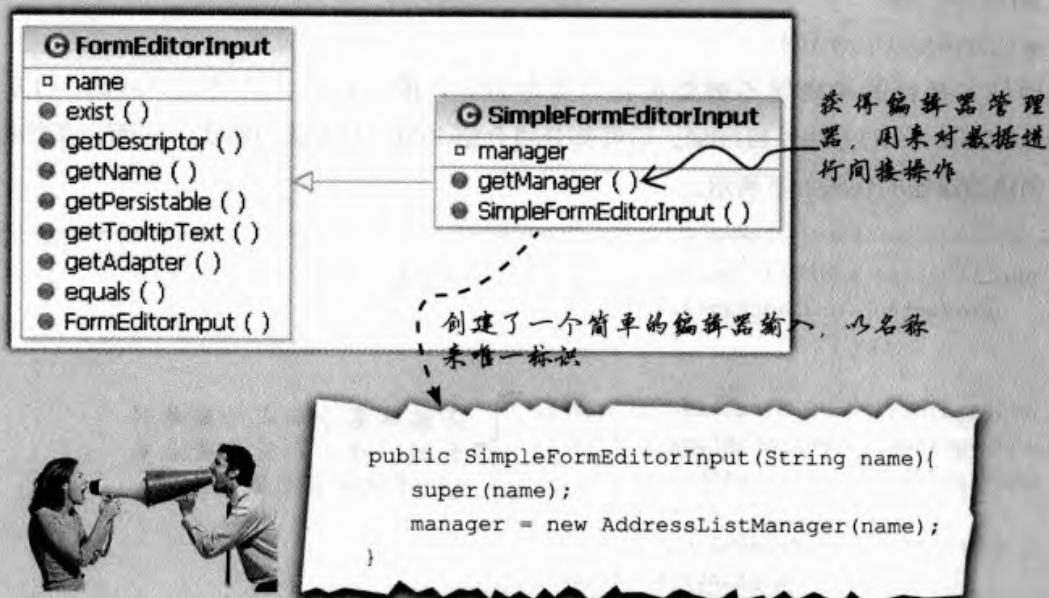


图13-6 FormEditorInput和SimpleFormEditorInput

※ 注意 ※

这里FormEditorInput的equals()方法默认实现是当两个输入的name相同时就认为这两个输入等价。在SimpleFormEditorInput的实现中仍然这样认为。



13.3.4 关联编辑器与编辑器输入

创建完编辑器输入后，在哪里完成编辑器和编辑器输入的关联呢？每个AddressItem都应该有其自己的编辑器输入，因而地址本插件编辑器中需要将AddressItem同编辑器输入一一对应。而且，编辑器同特定的编辑器输入结合应该发生在打开编辑器时。

本书在打开编辑器时结合编辑器输入同编辑器。在地址本视图（com.plugindev.addressbook.views.AddressView）中声明一个doubleClickAction的方法，该方法在双击地址本视图时创建编辑器输入，并将其同编辑器结合在一起。图13-7描述了创建双击操作的过程。

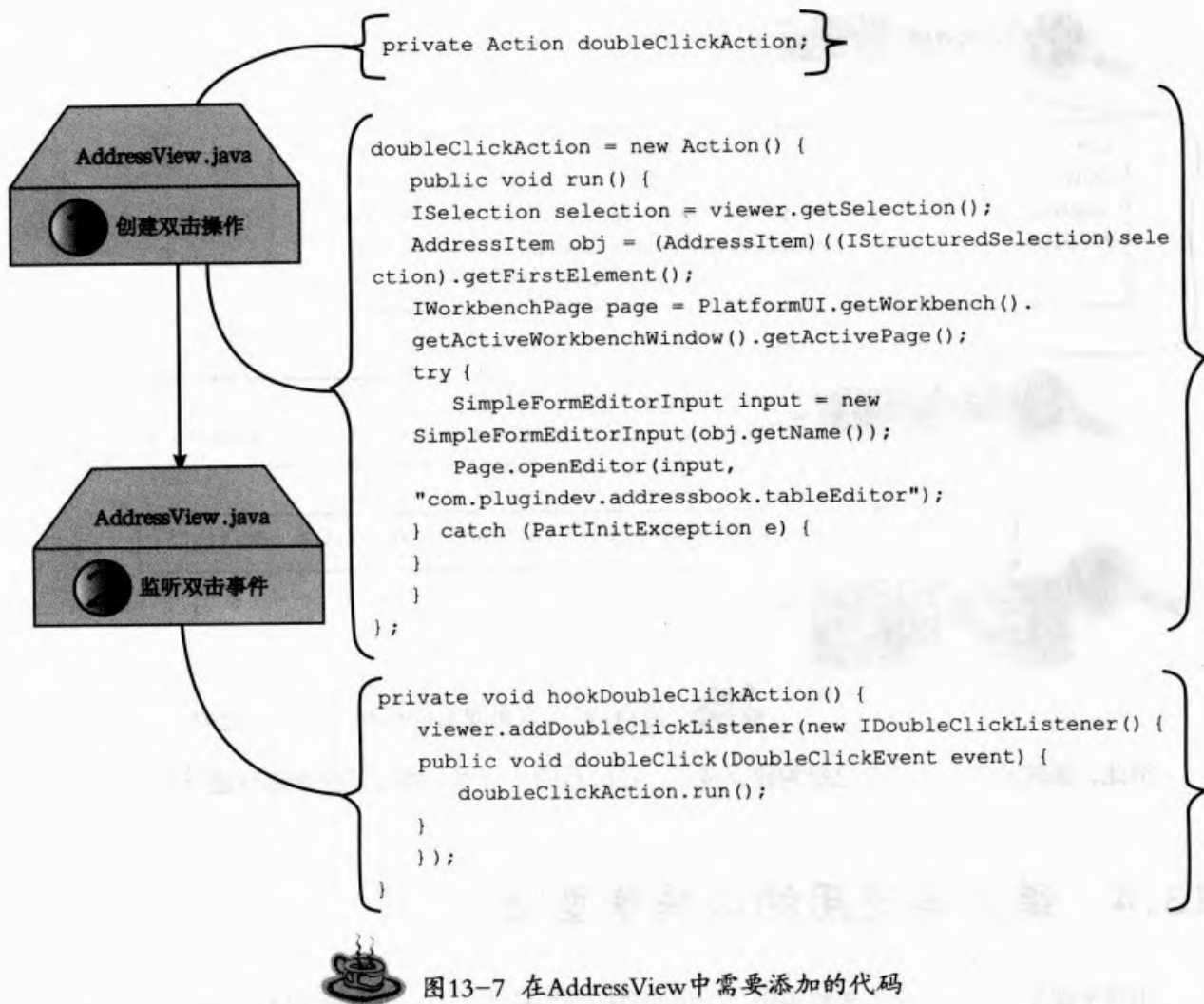
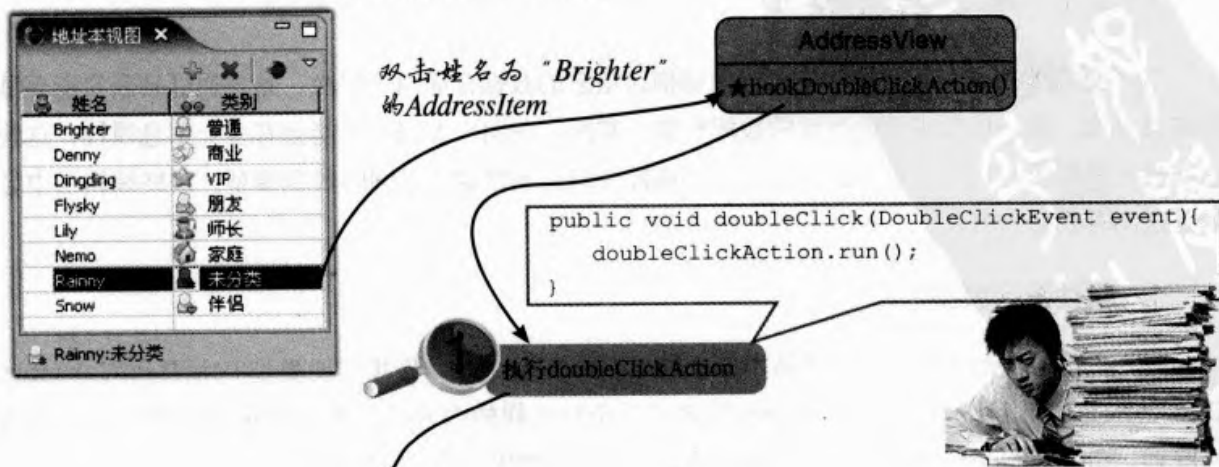


图13-7 在AddressView中需要添加的代码

由13-7可以看到，在地址本视图对双击事件提供支持的方式为首先在AddressView中创建双击操作，然后在hookDoubleClickAction()方法中为AddressView中的视图查看器添加doubleClickListener来监听双击事件的发生。

编辑器同编辑器输入的关联过程如图13-8所示。



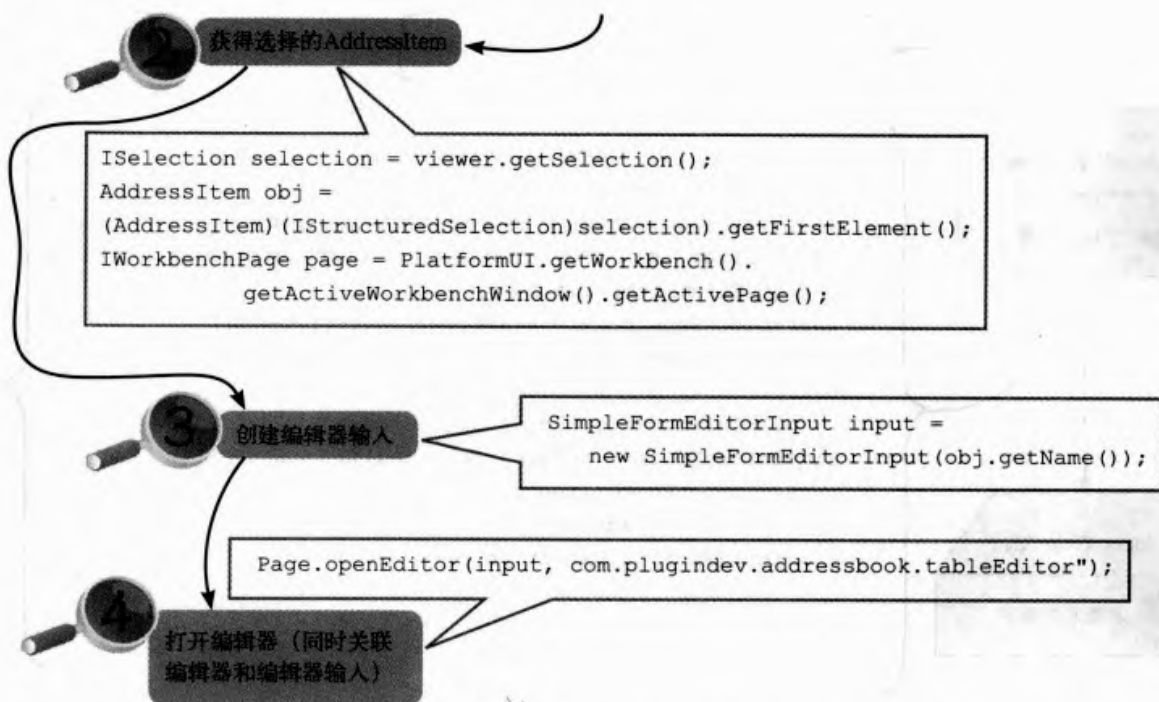


图13-8 关联编辑器和编辑器输入的过程

至此，编辑器的框架部分已经构建完成了。在以后的几节里，将对该框架部分进行填充。

13.4 编辑器使用的数据模型

相对于视图来说，地址本表格编辑器使用的数据模型较为复杂。可以分为以下三类。

- ★基本数据模型
- ★数据管理模型
- ★文件映像模型

13.4.1 基本数据模型

为了更好地表达层次化的信息，这里将描述信息的数据分成两个层次，首先通过列表将所要显示的信息分类。第一层用来描述类别的数据模型。其次，在同一类别中可能会有多个信息属性，这些信息属性可能要求不同的显示格式，因而还应该针对信息属性定义不同的数据模型。最终编辑器中显示的是这两种模型的组合。

1. 属性数据模型

如图13-9所示的类图介绍了基本类型接口IItemContents及其实现类型TextItemContents和ChoiceItemContents。IItemContents规定了无论什么样的内容项，都需要包含两个元素，键名称(key)和值(value)。在此规定，所有元素的键名称都用字符串(String)来表示。

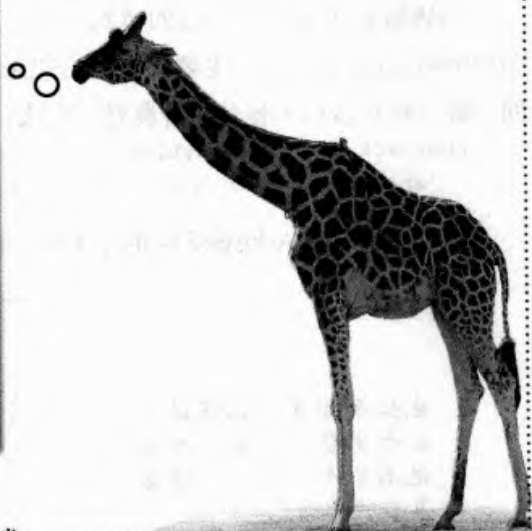
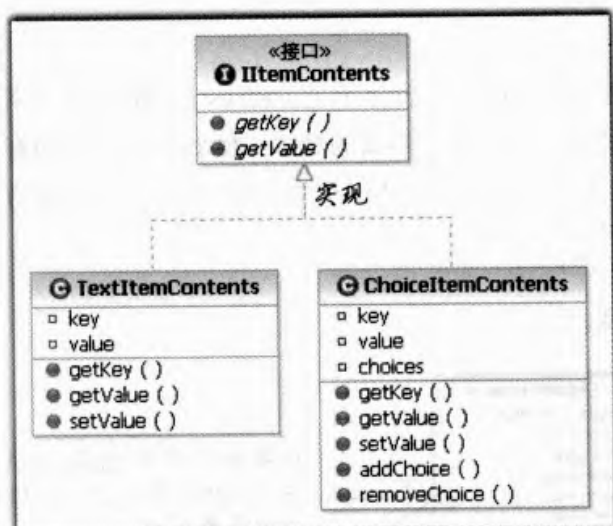


图13-9 基本类型IItemContents及其实现类

在地址本插件编辑器中，提供了两种类型的内容项（如需要还可以添加），一种是文本元素内容项，另外一种选择元素内容项。文本元素内容项的值也由字符串来表示，而选择元素内容项比较复杂。就最终界面的效果来看，每个选择内容项都将由一个或多个Button来表示，而每个Button都会有名称和其代表的整型值。图13-10描述了显示与模型之间的关系。

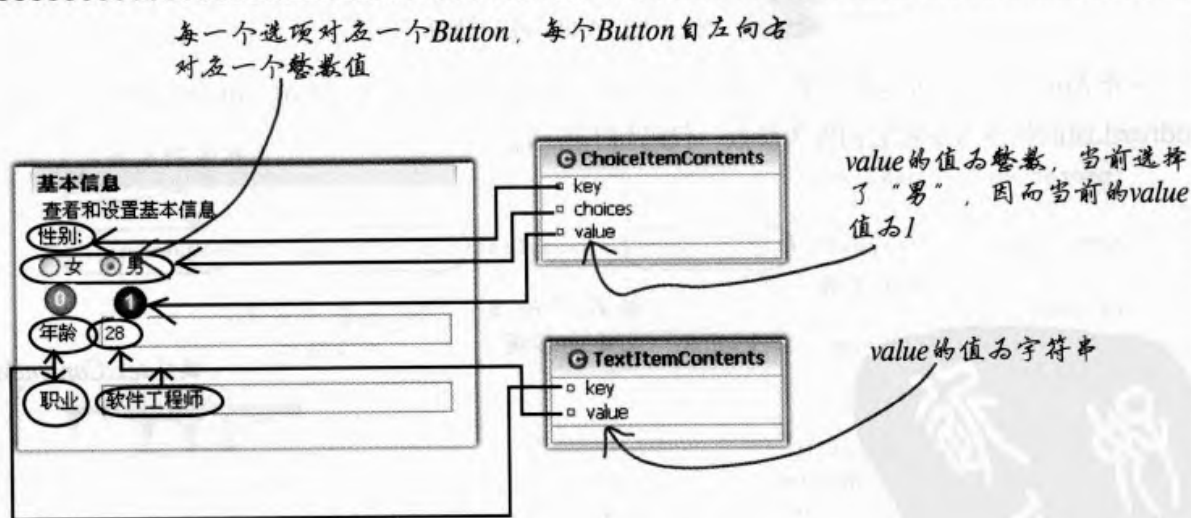


图13-10 选择项和文本框

图13-10中的“基本信息”选段中，包含有三个属性内容，第一个属性名称为“性别”，第二个为“年龄”，第三个为“职业”。它们的名称便是IItemContents实例中的key。

在ChoiceItemContents类中，addChoice()和removeChoice()用来为选择项内容提供或删除选项，如图13-10中的“女”和“男”。value的值记录了当前选中的选项，choices保存了所有选项的列表。

2. 列表数据模型

列表数据模型记录信息的类别，列表数据模型图13-11显示了AddressList的继承关系。其中，AddressList是抽象类，主要的方法由它来实现。但是对于不同的具体AddressList，所拥有的内容不同。统一针对接口编程是一种良好的方式，因而，在AddressList类中设置了如下两个抽象方法。

```
abstract public ArrayList getStringKeys();
abstract public Map getChoiceKeysMap();
```

这两个方法在AddressList的子类中实现。



图13-11 AddressList的继承关系

一个AddressList可能包含若干个ITextItemContents和IChoiceItemContents的组合，因而，AddressList的构造方法将它们作为参数，代码如下所示。

```
protected AddressList(
    String name, ArrayList stringItems, ArrayList choiceItems, int sequence) {
    this.name = name;
    this.sequence = sequence;

    if(textContentMap == null){
        textContentMap = new HashMap<String, TextItemContents>();
        for(int i = 0; i < stringItems.size(); i++){
            TextItemContents item = (TextItemContents)stringItems.get(i);
            textContentMap.put(item.getKey(), item);
        }
    }

    if(choiceContentMap == null){
        choiceContentMap = new HashMap<String, ChoiceItemContents>();
        for(int i = 0; i < choiceItems.size(); i++){
            ChoiceItemContents item = (ChoiceItemContents)choiceItems.get(i);
            choiceContentMap.put(item.getKey(), item);
        }
    }
}
```

表示该列表的优先级

列表名称

保存了所有文本内容项

保存了所有文本选择项

填充TextContentMap

```

ChoiceItemContents item = (ChoiceItemContents)choiceItems.get(i);
    choiceContentMap.put(item.getKey(), item);
}
}

```

填充choiceContentMap

在AddressList中还定义了textContentMap和choiceContentMap两个Map，声明代码如下所示，分别记录键名称和相应的ItemContents的对应关系，它们在构造函数中被实例化。可以通过getTextContentMap()和getChoiceContentMap()两个getter方法来得到它们。

```

protected Map<String, TextItemContents> textContentMap;
protected Map<String, ChoiceItemContents> choiceContentMap;

```

由文本项的键名称获得文本项的相关信息

由选择项的键名称获得选择项的相关信息

在实现的AddressList的三个子列表里，每个子类包含了一些默认的ItemContents实例（键值对），在程序启动之后，它们将分别存储在各自的stringKeys和choiceKeysMap中。最终的用户很可能希望，当确定了列表之后，列表包含的键名称是固定的，即一个特定的列表对应这一组特定的键名称，因此，代码中各自的stringKeys和choiceKeysMap都被设置为静态值。以BasicAddressList为例，该类中对stringKeys和choiceKeysMap的声明如下所示。

```

public static ArrayList<String> stringKeys;
public static Map<String, Object[]> choiceKeysMap;

```

记录所包含的TextContentsItem的键名称

记录所包含的ChoiceContentsItem的键名称和其每个整数值所代表的含义

BasicAddressList构造方法的任务便是继承父类构造方法，并且填充这两个静态变量的值，代码如下所示。

```

public BasicAddressList(
String name, ArrayList stringItems, ArrayList choiceItems, int sequence) {
    super(name, stringItems, choiceItems, sequence);
    if(stringKeys == null){
        stringKeys = new ArrayList<String>();
        for(int i = 0; i < stringItems.size(); i++){
            TextItemContents item = (TextItemContents)stringItems.get(i);
            stringKeys.add(item.getKey());
        }
    }
}

```

填充stringKeys以记录现有的TextContentsItem（标签+文本框）

```

    if(choiceKeysMap == null){
        choiceKeysMap = new HashMap<String, Object[]>();
        for(int i = 0; i < choiceItems.size(); i++){
            ChoiceItemContents item = (ChoiceItemContents)choiceItems.get(i);
            choiceKeysMap.put(item.getKey(), item.getChoices());
        }
    }
}

```

填充choiceKeysMap以记录现有的ChoiceContentsItem（标签+文本框）

其他两个地址列表子类同BasicAddressList类似, 可以通过以下两种方法来获得某个地址列表的键名称。

★当不清楚地址列表实例的具体子类型时, 使用AddressList中定义的抽象方法getStringKeys(), getChoiceKeysMap();

★若需要获得特定地址列表中包含的键名称, 如获得BasicAddressList的键名称, 使用BasicAddressList.stringKeys和BasicAddressList.choiceKeysMap。

默认每个列表定义的键名称和属性模型如表13-4所示(在AddressListManager中定义, 详见第13.4.2节“数据管理模型”), 可以根据需要进行添加和改动。

表13-4 列表默认设定的键名称和属性模型

列表(分类)名称	键名称	属性模型
基本信息 (BasicAddressList)	性别	CheckItemContents
	年龄	TextItemContents
	职业	TextItemContents
联系电话 (PhoneAddressList)	家庭电话	TextItemContents
	办公电话	TextItemContents
	手提电话	TextItemContents
	其他电话	TextItemContents
联系地址 (AreaAddressList)	家庭地址	TextItemContents
	办公地址	TextItemContents
	其他地址	TextItemContents

由表13-4可以看到, 联系电话和联系地址两个列表只包含TextItemContents, 因而未给它们设置choiceKeysMap。这两个列表中的getChoiceKeysMap()方法实现如下所示。

```
public Map getChoiceKeysMap() {
    return new HashMap();
}
```

列表数据模型同基本数据模型的关系如图13-12所示。

AddressList中的getIntValue()
和setIntValue()方法分别调用
ChoiceItemContents的getValue()
和setValue()方法

AddressList中的getStringValue()
和setStringValue()方法分别调用
TextItemContents的getValue()
和setValue()方法

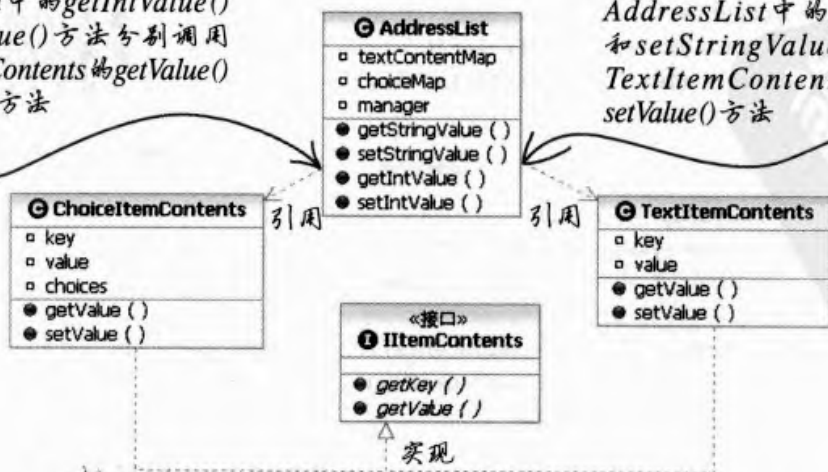
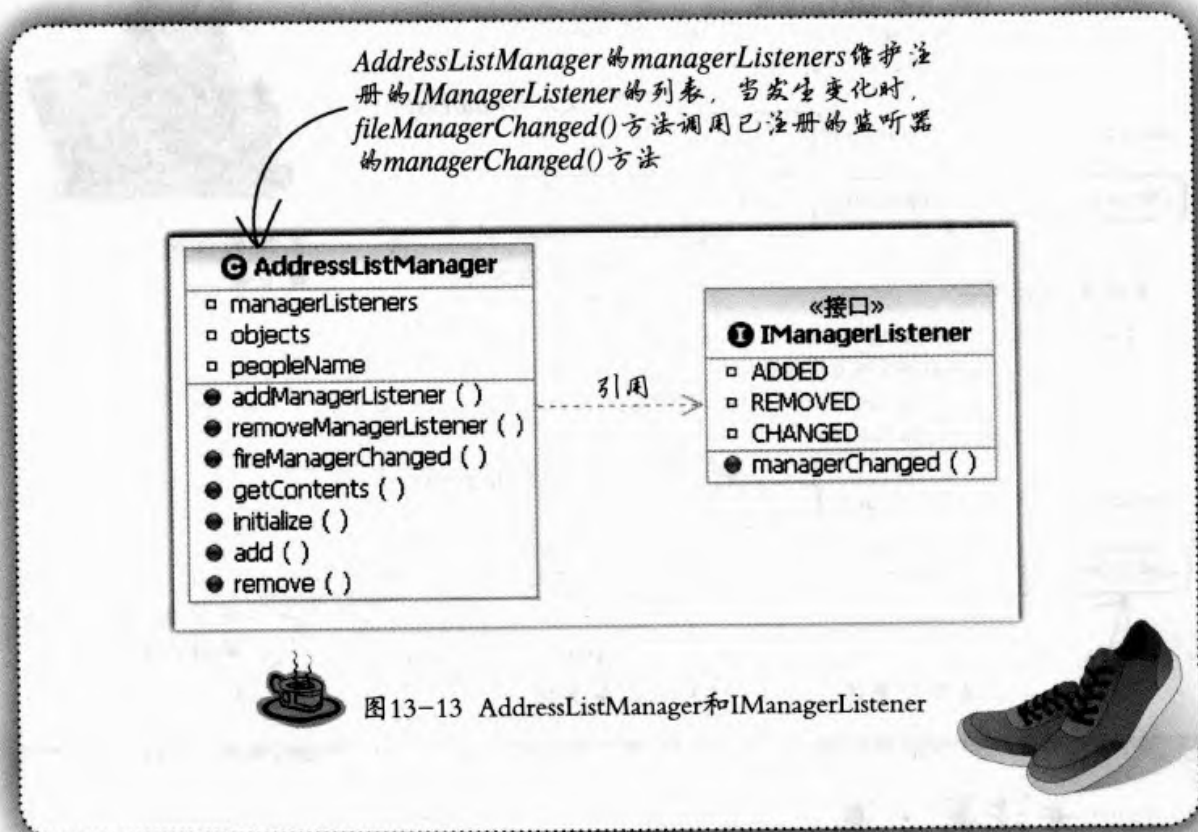


图13-12 列表数据模型通过基本数据模型设置和获得相应的值

13.4.2 数据管理模型

本节仅为地址本插件定义了AddressList的管理模型AddressListManager，它负责AddressList的初始化、增加、删除、事件监听以及后面要加入的保存、存储等功能。AddressListManager管理的粒度比较大，但作为示例程序，已经能够满足要求。

为了监听对AddressList的增加、删除、更改等事件，AddressListManager还需要维护一个监听器列表，以便在事件发生时通知在其中注册的监听器。对应的监听器所要实现的方法由IManagerListener接口定义。图13-13显示了AddressListManager和IManagerListener之间的关系。



一个AddressListManager对应一个地址本视图中的AddressItem，并且以AddressItem中的人名作为其标识。因此，AddressListManager的构造方法如下所示。

```
public AddressListManager(String name) {
    managerListeners = new ArrayList<IManagerListener>();
    this.peopleName = name;
    initialize();
}
```

initialize方法中创建了表13-4中的三个默认的AddressList实例。在本章后面的部分中，将完善initialize方法，使其能够从文件中读入数据。

AddressListManager需要响应以下三种事件。

★更改事件

★增加AddressList事件

★删除AddressList事件

对于增加和删除AddressList的事件，AddressListManager采取灵活的方式处理，它根据需要设置了一个布尔值notify，用来指示要不要通知注册的监听器。

任何对AddressList的更改操作都需要通知AddressListManager，因此需要修改AddressList类中所有的setter方法（这也是需要在AddressList中维护一个AddressListManager实例的原因），代码如下所示。

```
public void setStringValue(String key, String value) {
    // TODO 自动生成方法存根
    TextItemContents item = textContentMap.get(key);
    item.setValue(value);
    manager.fireManagerChanged(this,
        IManagerListener.CHANGED, Messages.ITEM_TYPE_TEXT, key);
}

public void setIntValue(String key, int value) {
    // TODO 自动生成方法存根
    ChoiceItemContents item = choiceContentMap.get(key);
    item.setValue(value);
    manager.fireManagerChanged(this,
        IManagerListener.CHANGED, Messages.ITEM_TYPE_BUTTON, key);
}
```

修改的AddressList

资源更改事件

修改的为文本类型


更改的值所对应的键名称

修改的AddressList

资源更改事件

修改的为选择按钮类型


更改的值所对应的键名称



✦ 注意： ✦

fireManagerChanged方法中包含了4个参数，这同IManagerListener中定义的managerChanged()方法是一致的。4个参数有点多，但还可以接受，对于你例程序已经足够了。如果对程序代码要求更高的可复用性，就需要创建一个Event事件来封装这些参数。同样的道理，对于AddressList的构造方法中引用的参数也可以再封装。

地址本插件于编辑器相关的字符串全部保存在com.plugindev.addressbook.util.Messages中，可以查看光盘中的源代码来了解这些字符串。



13.4.3 文件映像模型

文件映像模型负责将模型存储到文件中。地址本插件选择以XML的方式存储信息，因为XML格式的文件能够很方便地记录层次结构，便于信息的分级显示。

在第12章“视图”中使用了Eclipse中的org.eclipse.ui.XMLMemento来保存AddressItem。为了简便起见，编辑器将使用类似的机制来保存，但是XMLMemento本身具有一些限制。在对存储提供之前，将其复制到地址本插件里，重构其名称为MYXMLMemento，以方便修改。XMLMemento可以从XML文件中提取出DOM（Document Object Model，文本对象模型），因此，虽然Eclipse为它规定了许多限制，仍然可以使用它作为自己定义的XML文件解析器（有关XMLMemento的限制及更多相关信息请参阅拙作“XML对象的相关分析”一文。见网址http://www.blogjava.net/nemo-zhp/archive/2007/04/26/Eclipse_XMLMemento.html）。

13.5 编辑器页面

本节将创建如下三个编辑器页面，这三个页面都需要继承FormPage类。

- ★ “编辑” 页
- ★ “分页预览” 页
- ★ “源代码” 页

其中，“编辑”页作为主要编辑信息项的页面，而“分页预览”页将不同类别的信息列表在不同的页面上显示（但是不可编辑），“源代码”页显示存储在文件中的源代码。

FormPage在org.eclipse.ui.forms.editor中定义，它的继承体系如图13-14所示。

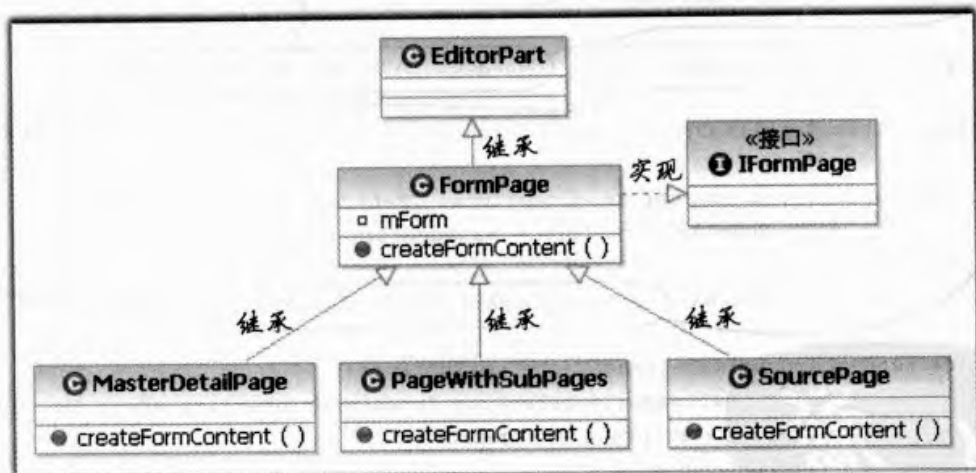


图13-14 FormPage的继承体系

FormPage是构建FormEditor的页面的一个基本类型，所有加入FormEditor的页面都必须继承FormPage。在FormPage类中，保存了内部类PageForm的实例mform，PageForm是ManagedForm的子类，为方法createFormContent(ManagedForm)提供参数。FormEditor的子类型可以通过重写createFormContent(ManagedForm)来构建特定页面的内容。

FormPage本身也是按照懒加载的原则创建的。创建表格内容的操作（由方法createFormContent()来完成）可以在编辑器被打开之后被执行，甚至可以在编辑器声明周期中不被执行，只要用户不需要查看

该页面的显示内容。

在图13-14中，FormPage的三个子类分别描述了三种类型的页面。

13.5.1 “编辑”页

“编辑”页使用了Master-Detail（主-从）风格。在Master区域列出了使用的AddressList，Detail区域列出了包含在该AddressList的所有属性。最终显示效果如图13-15所示。



```
public class MasterDetailsPage extends FormPage {
    private ScrolledPropertiesBlock block;
    public MasterDetailsPage(FormEditor editor) {
        super(editor, "masterDetail", Messages.PAGE_NAME_MASTERDETAIL);
        block = new ScrolledPropertiesBlock(this);
        protected void createFormContent(final IManagedForm managedForm) {
            final ScrolledForm form = managedForm.getForm();
            form.setText(Messages.PAGE_TITLE_MASTERDETAIL);
            form.setBackgroundImage(ImageCache.getInstance().
                getImage(ImageKeys.getImageDescriptor(ImageKeys.IMG_FORM_BG)));
            block.createContent(managedForm);
        }
    }
}
```



图13-15 Master-Detail风格页面的显示效果

Master-Details页面创建好后，需要为Master-Detail页面创建内容。org.eclipse.ui.forms中的抽象类MasterDetailsBlock可以达到此目的，这里创建ScrolledPropertiesBlock来实现这些方法。图13-16显示了MasterDetailsBlock和ScrolledPropertiesBlock中包含的方法。

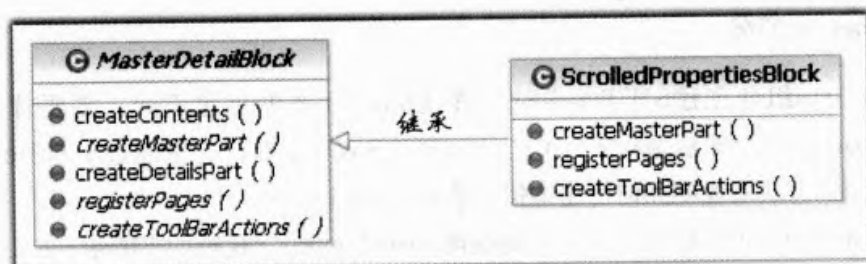


图13-16 ScrolledPropertiesBlock类的继承关系

MasterDetailsBlock类采用注册页面的方式来构建主-从块。registerPages()方法用来为具体的从属页面注册所允许的数据模型，从而建立起从属页面和数据模型的一一对应关系。从属页面也是采用注册方式在createDetailsPart()方法中注册到主页面中。

图13-17演示了MasterDetailsBlock创建“编辑”页面的过程。

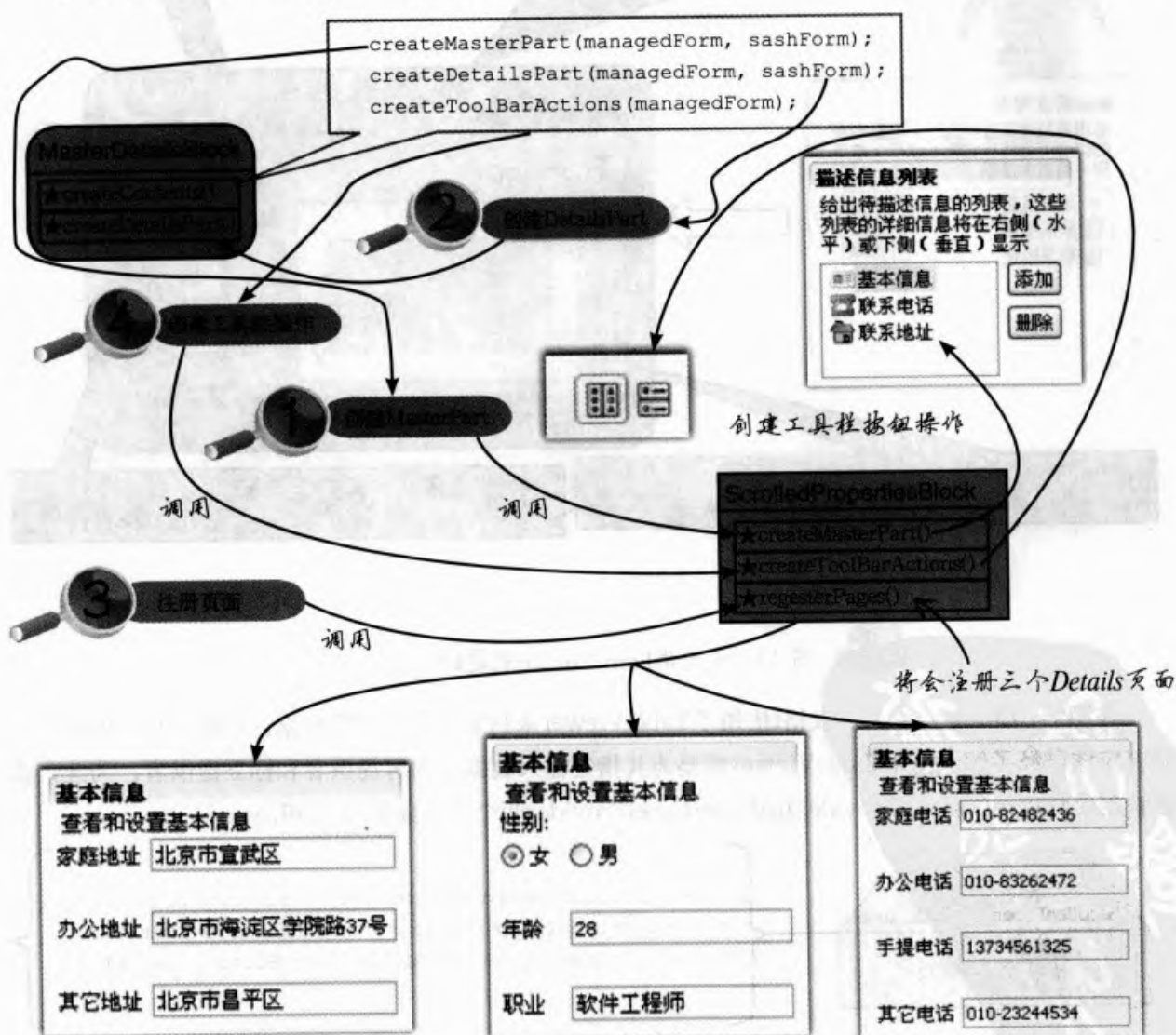


图13-17 创建“编辑”页面的过程

1. 构建Master部分

Eclipse的Form组件创建SWT元素时同普通SWT元素的创建不同，需要首先得到Form工具包（FormToolkit），然后通过Form工具包创建SWT元素。这里通过ScrolledProperties的createMasterDetail()方法来说明如何构建编辑器的Master部分。

在createMasterDetail()方法中，首先需要获得FormToolkit，代码如下所示。

```
FormToolkit toolkit = managedForm.getToolkit();
```

然后，使用FormToolkit创建了如图13-18的窗口部件，图中列出了创建各个子部件所用的方法。

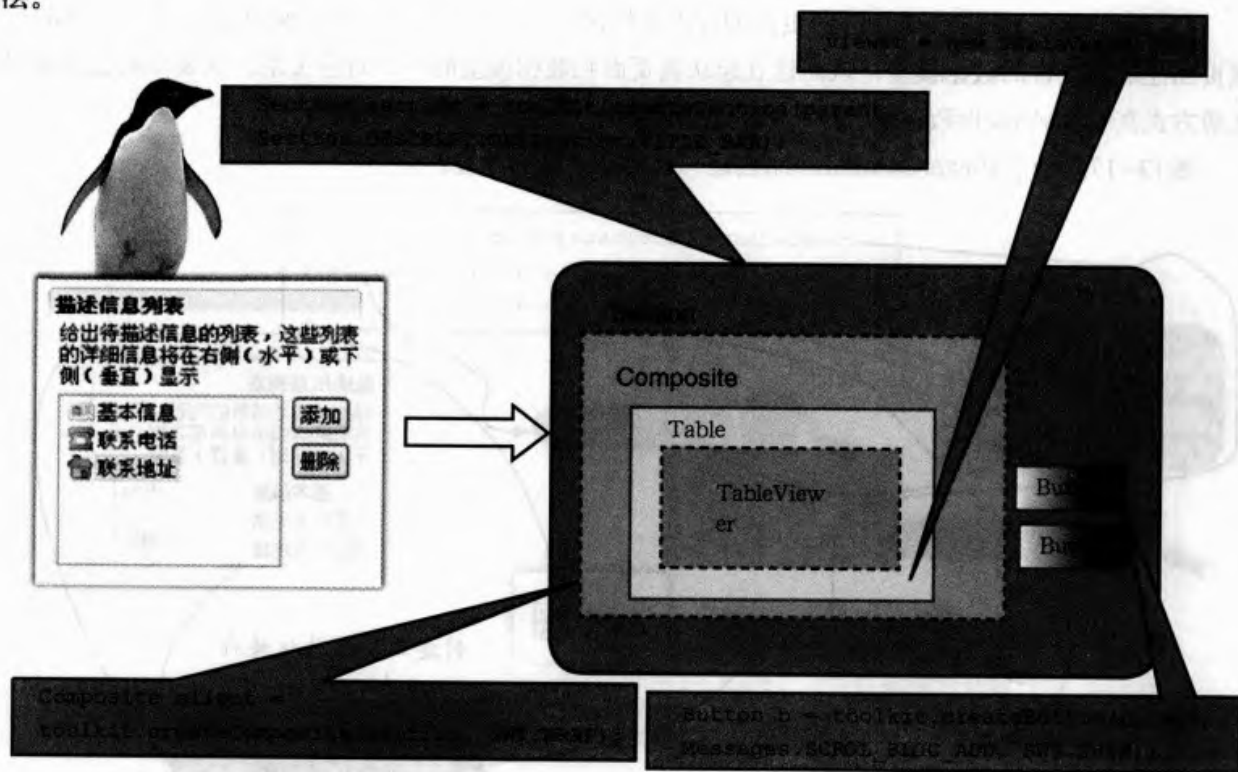
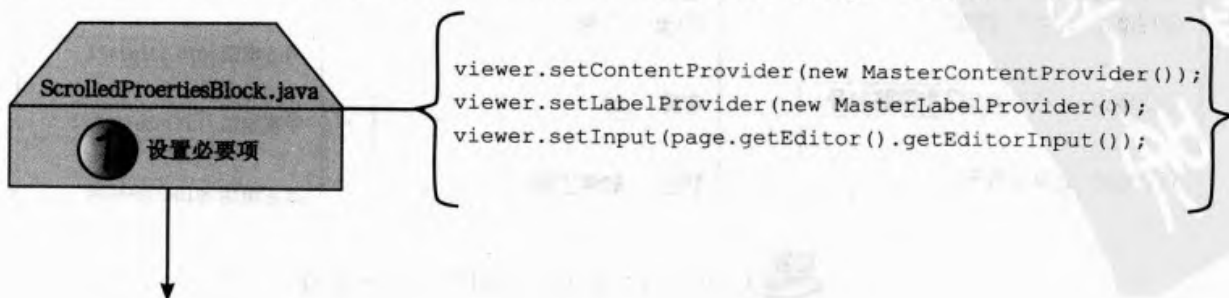


图 13-18 使用FormToolkit创建UI元素

在图13-18中可以看到，表格t使用了TableViewer来封装表格使用的数据。回顾第12章的内容，读者应该已经了解，使用TableViewer需要为其指定输入数据、内容提供者和标签提供者。为此，需要新创建MasterContentProvider和MasterLabelProvider两个类，如图13-19所示。



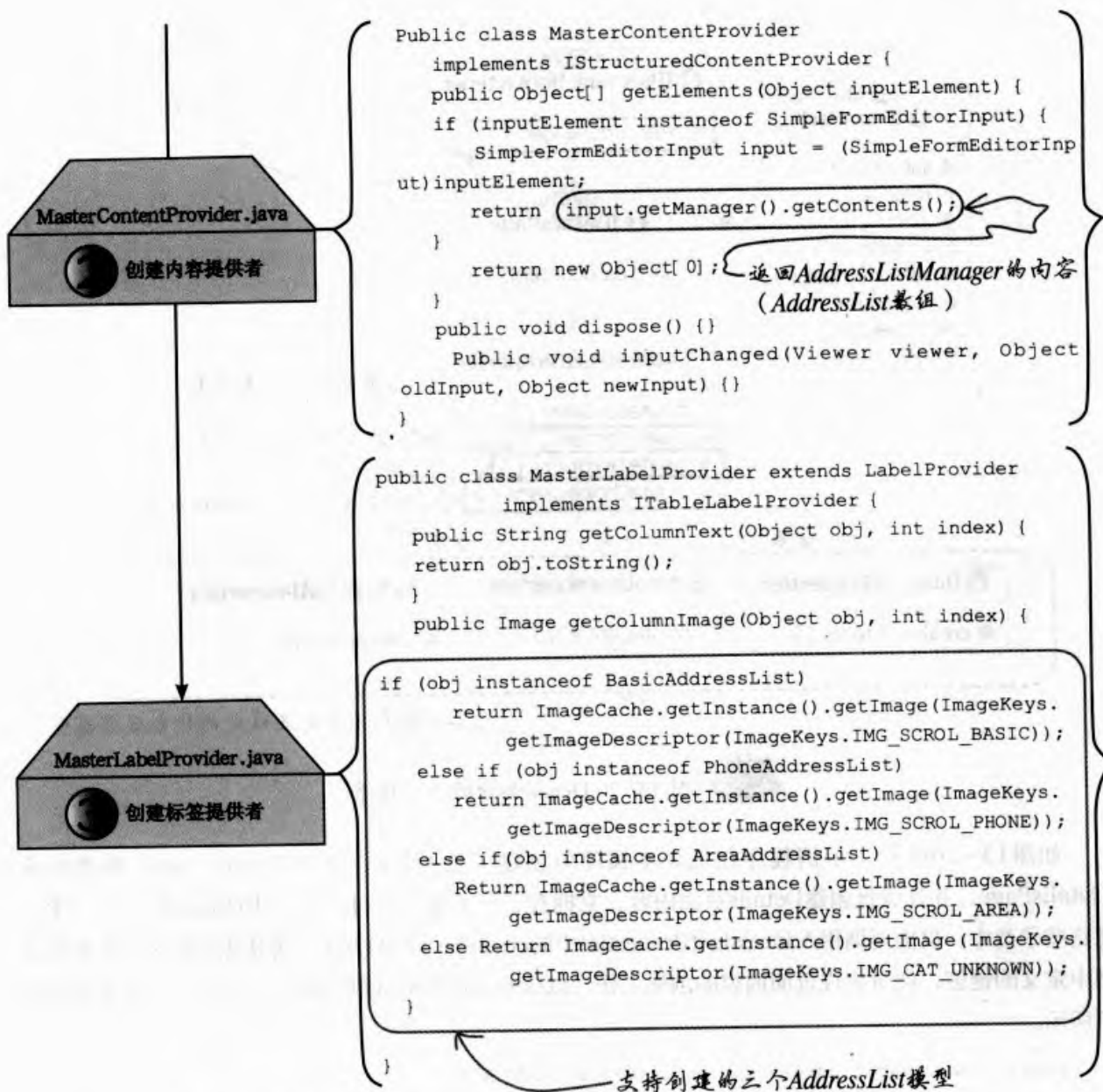


图13-19 为Master部分中的Table初始化表格查看器的步骤

创建完ContentProvider和LableProvider之后，在Master区域将会显示如图13-18中左侧所示的界面，界面中使用的图标在光盘本章项目源代码目录icons/editor中可以找到。

2. 构建Details部分

构建Details部分需要实现IDetailsPage接口。IDetailsPage接口在IFormPart和IPartSelectionListener的基础上增加了createContents()方法。Details部分作为一个特定的表格部分，它需要拥有管理表格生命周期的方法，这些方法通过继承IFormPart接口定义。另外，由于Details部分需要监听Master部分并根据Master部分的选择做出改变，因而需要继承IPartSelectionListener接口。Details部分的继承体系如图13-20所示。

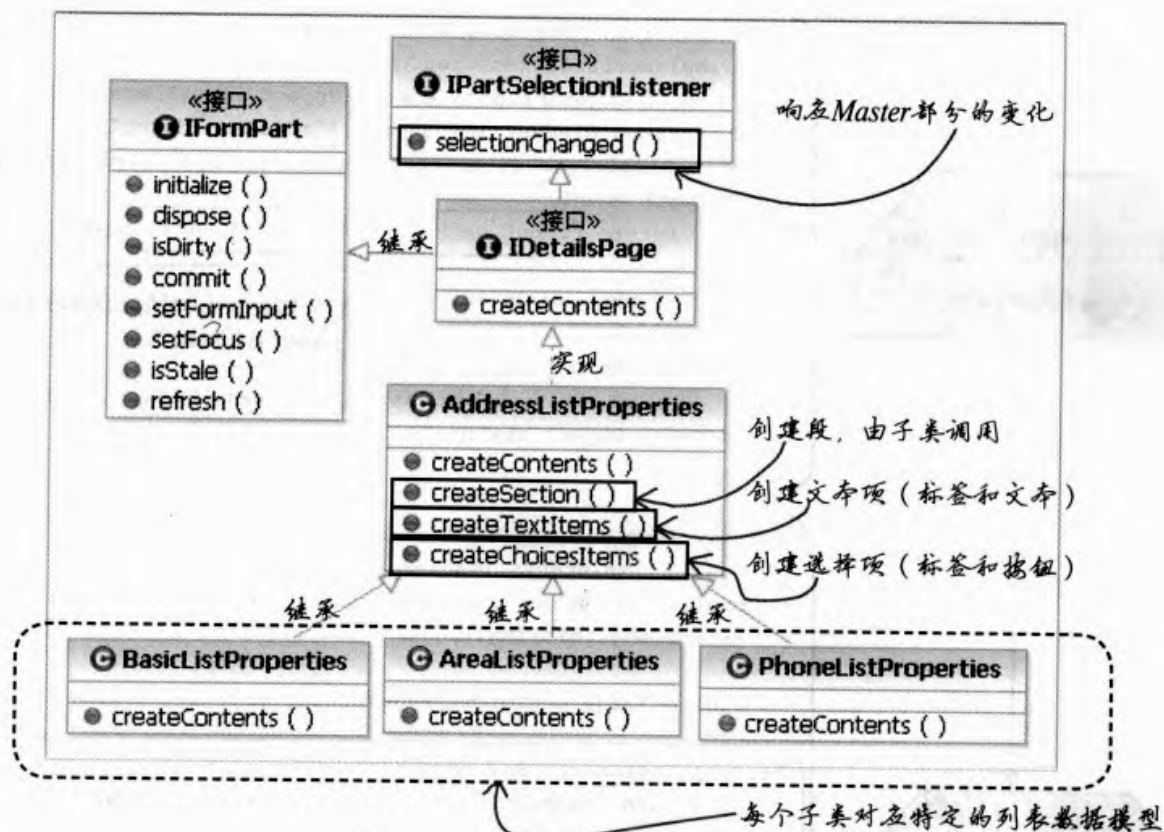


图 13-20 Details部分的继承体系

如图13-20所示，示例程序在地址本插件中创建了类AddressListProperties，该类继承IDetailsPage，并且成批创建Details页面内容。它拥有三个子类，分别对应AddressList的三个子类。在这些子类中，仅需要调用AddressListProperties中的createSection()方法，为其传递相应列表数据模型中定义的键值，便可完成页面内容的创建工作。以BasicAddressListProperties为例，其实现代码如下所示。

```
public class BasicAddressListProperties extends AddressListProperties {
```

```
    public BasicAddressListProperties() {
        super();
```

调用AddressListProperties的
createContents()方法，完成基
本框架的创建

```
    public void createContents(Composite parent) {
```

```
        super.createContents(parent);
```

```
        this.createSection(parent, BasicAddressList.stringKeys,
            BasicAddressList.choiceKeysMap);
```

为createSection()方
法传递Basic列表模
型中定义的键名

```
    }
}
```

Details部分也是采用懒式加载的模式创建。为了帮助理解这个过程，在此以BasicAddressList的创建和加载为例，创建的过程如图13-21所示。

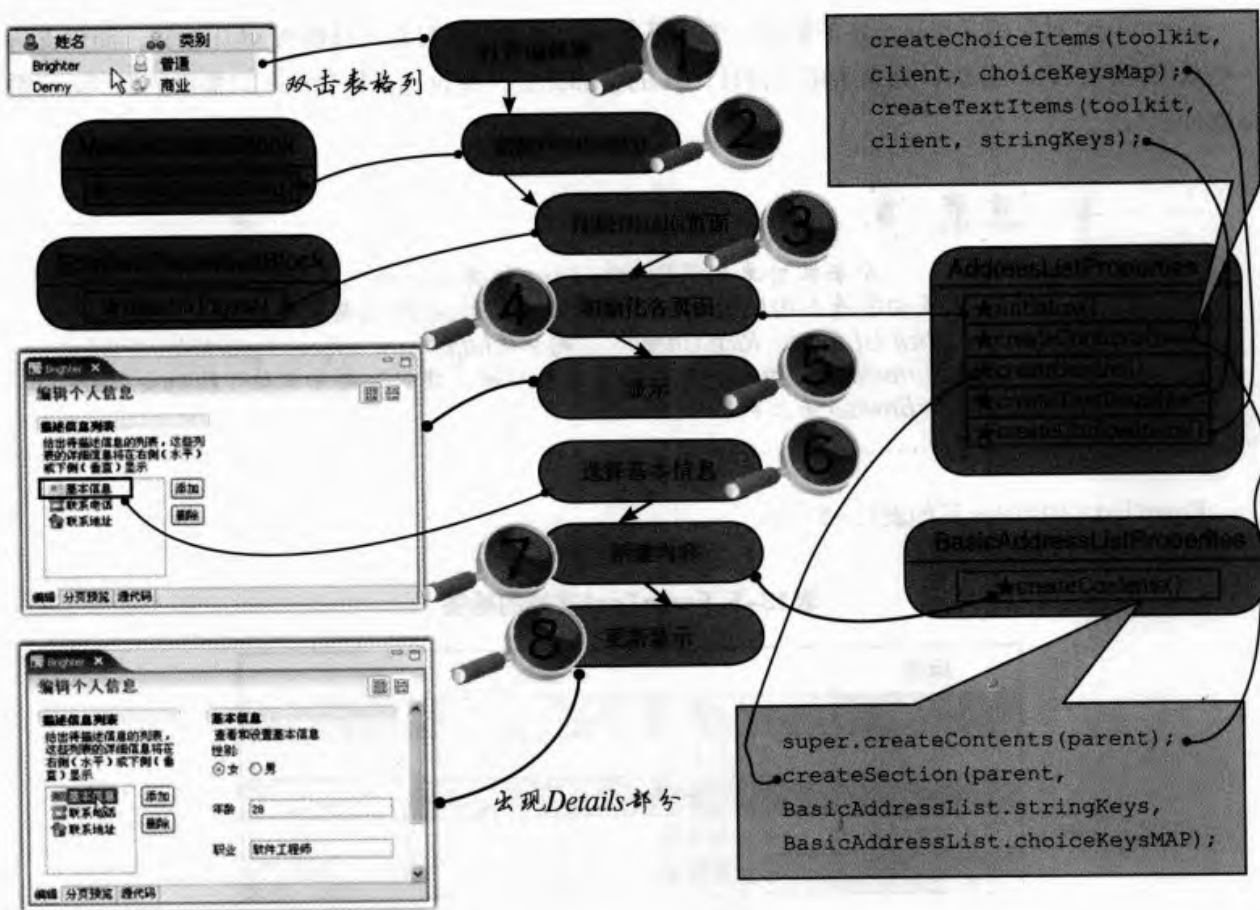
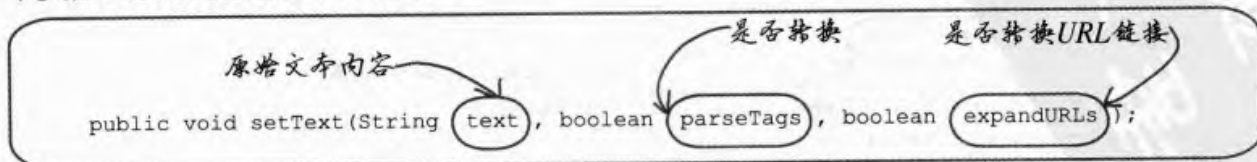


图13-21 懒式加载Details部分的过程

图13-21中，前1~5步为注册Details页面的过程，这个过程在打开编辑器操作执行时发生。在这个过程中，IDetails类的实例被创建（调用了AddressListProperties类的initialize()方法），但是并未创建显示的内容。只有当用户第一次用鼠标选择Master部分的“基本信息”的事件发生时，createPartControl()方法被激活，BasicAddressListProperties部分的内容才第一次被加载，加载的过程如图13-21的步骤6、7、8所示。

另外两个IDetails页面PhoneAddressListProperties和AreaAddressListProperties的创建的懒式加载过程同BasicAddressListProperties类似，在此不再一一说明。

为了显示Eclipse Form功能的强大，示例程序还在AddressListProperties中创建了FormText，FormText可以模仿HTML页面对简单HTML代码进行解析。在FormText类中，方法setText具有三个参数，它们的含义分别如下所示。



parseTags标志位如果为真值，则将会自动解析原始文本中可识别的XML标签；而如果expandURLs为真值，则在显示文本时会自动转换以“http://”标签开头的文本为超链接。

FormText对原始文本有一定的限制,它刚开始设计时就没有打算支持所有XML标签,而仅支持一些最常用的HTML标签,如果希望支持HTML的全部标签,建议选择Eclipse UI提供的Browser窗体部件。



注意

本书仅简单介绍Eclipse Form组件的使用,了解Eclipse Form的更多内容请参阅Eclipse官方网站有关Eclipse Form的文档“Eclipse Forms: Rich UI for the Rich Client”,网址为<http://www.eclipse.org/articles/Article-Forms/article.html>。在这篇文章最后的“建议”部分介绍了Eclipse Form和Browser的区别。

FormText支持的标签如表13-5所示。

表13-5 FormText支持的标签

标签	描述
<p>	定义段落
	定义列表项
	显示图像
<a>	显示超链接
	使用粗体字
 	强制换行
	可以为其中的内容设置颜色和字体
<control>	在文本中放置一个控件(版本3.1中加入)



在AddressListProperties类的createSection()方法中加入如下代码。

```
FormText rtext = toolkit.createFormText(parent, true);
```

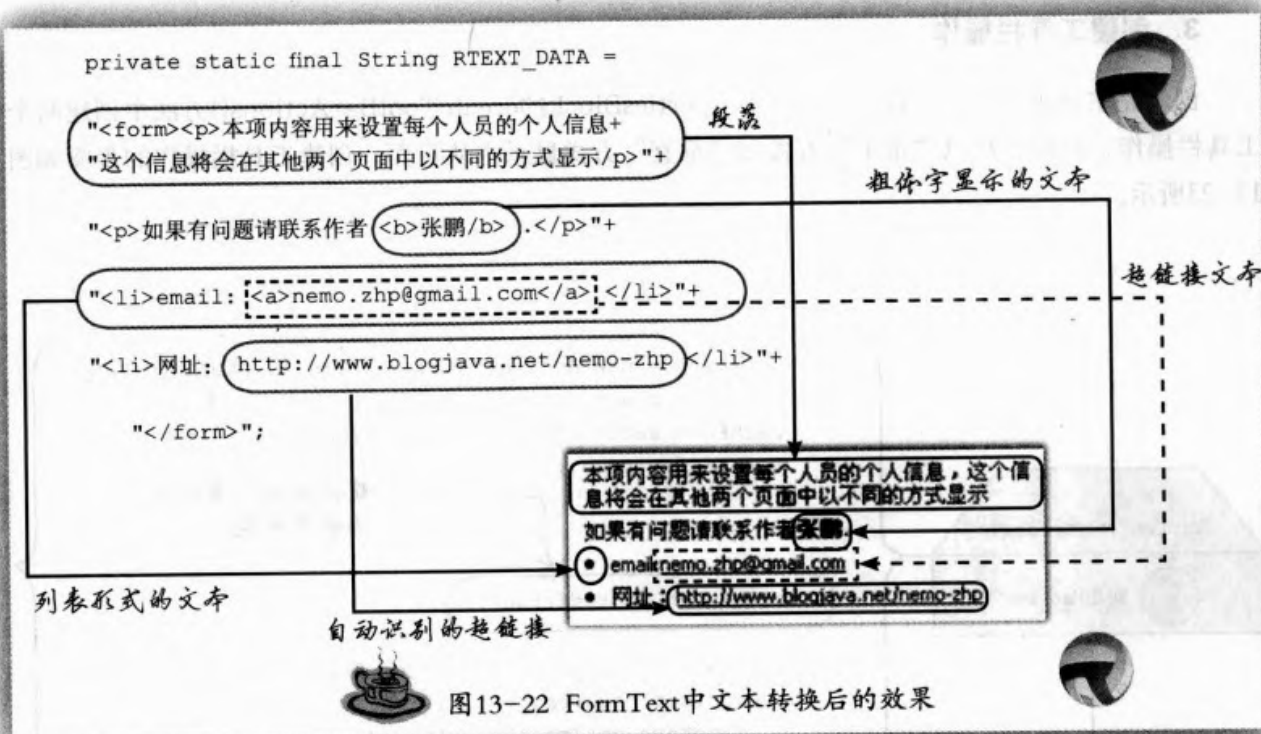
创建FormText

```
rtext.setText(RTEXT_DATA, true, true);
```

记录了原始文本的字符串

```
td = new TableWrapData(TableWrapData.FILL, TableWrapData.TOP);
td.grabHorizontal = true;
rtext.setLayoutData(td);
```

转换后的显示效果如图13-22所示。



在图13-22中，列表项Email地址是通过标签“<a>”转换的，而网址则是自动转换的，FormText的setText()方法将根据前缀“http://”自动识别。



3. 创建工具栏操作

创建完主从页面之后，在ScrolledPropertiesBlock的createToolBarActions()方法中创建两个工具栏操作，分别支持以“水平”方式和“垂直”方式显示主从页面。创建工具栏操作的步骤如图13-23所示。

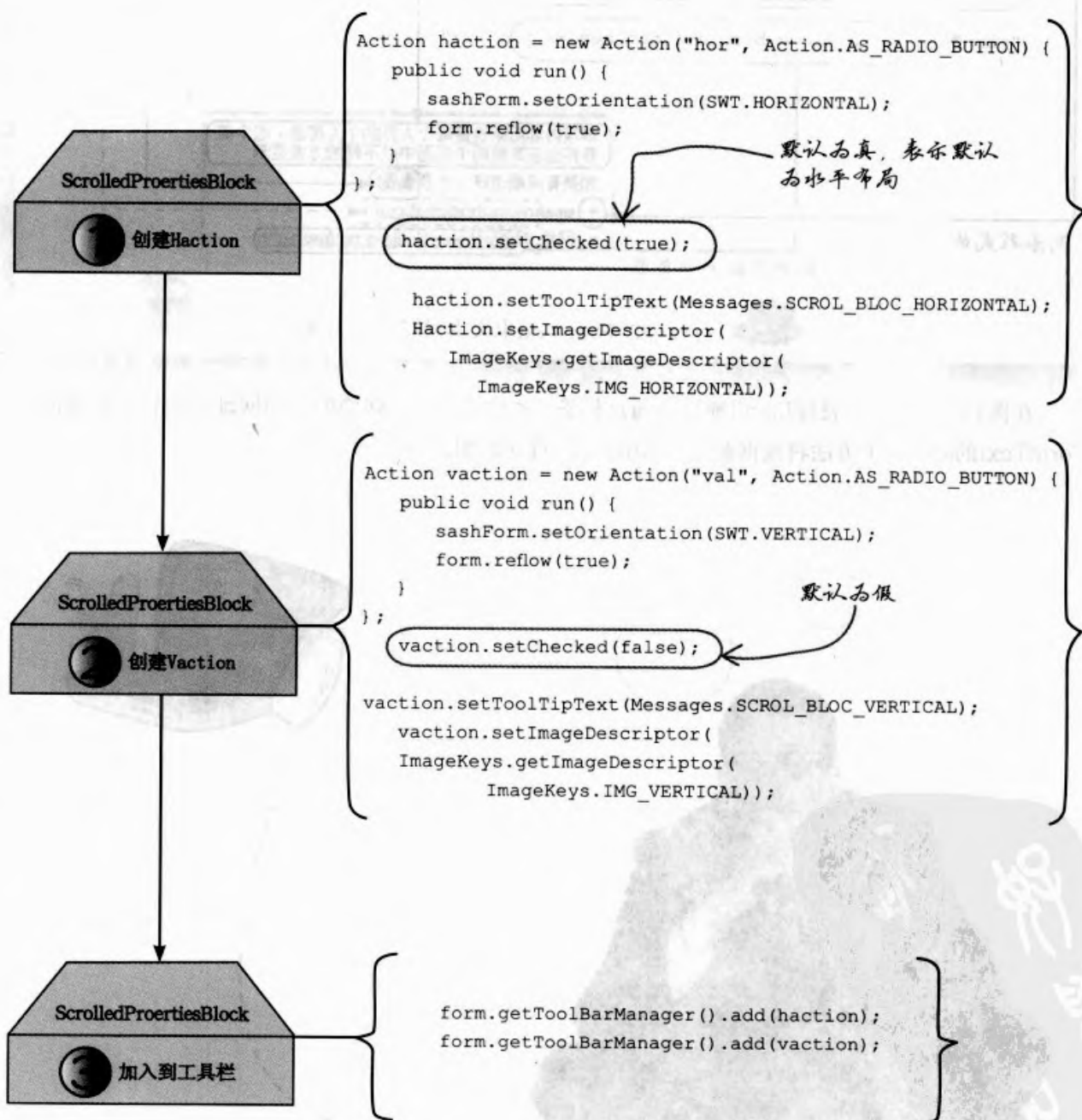


图13-23 为工具栏加入操作的步骤

加入工具栏操作后，选择不同的排列方式，“编辑”页面将按不同的方式排列如图13-24所示。

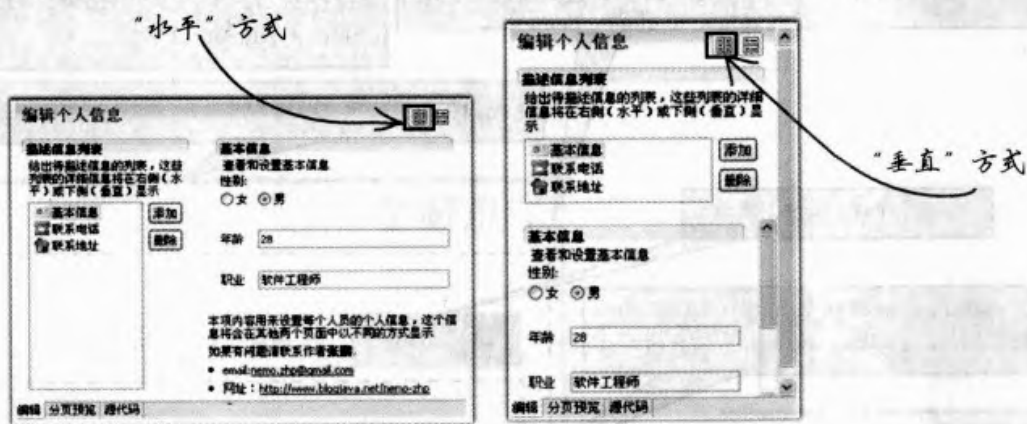


图13-24 编辑页面按不同方式排列的效果图

至此，已创建完成“编辑”页面。

13.5.2 “分页预览”页

“分页预览”页的功能是将“编辑”页面编辑的信息按照AddressList的个数分页，每页显示相应的AddressList的属性信息。在下一节“响应编辑器更改”中，示例程序将增加“分页预览”页的功能，使其能够支持动态响应AddressList模型的变化。

如图13-25所示为创建该页的过程代码。图中“分页预览”页创建了三个子页面，分别对应着“编辑”页面Master部分中的三个AddressList列表项。可能读者会想，如果该页面中的子页随着列表项的改变而改变就好了。那就通过方法createTabs()实现这个愿望！代码如下所示（在第15章“对话框和向导”中，还会看到子页随着列表项的增加而增加）。

```
private void createTabs(FormToolkit toolkit) {
    Object[] contents = ((SimpleFormEditorInput)editor.getEditorInput()).
    getManager().getContents();
    for(int i = 0; i < contents.length; i++)
        createTab(toolkit, contents[i].toString(), "");
}
```

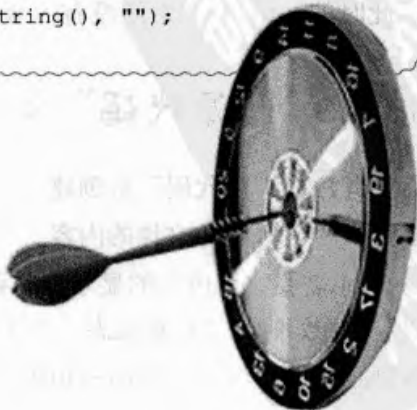




图13-25 “分页预览”页的创建过程

Section需要实现内容的显示和隐藏功能,如图13-26步骤 ② 所示。“源代码”页中的“预览源代码”段初始时处于折叠状态,当用户用鼠标单击箭头(小手所指定的位置)后,段将展开,其中的内容显示在页面中。在图13-26中还可以看到,当鼠标移动到指定的toggle区域时,该区域的颜色会发生变化(步骤 ①)。

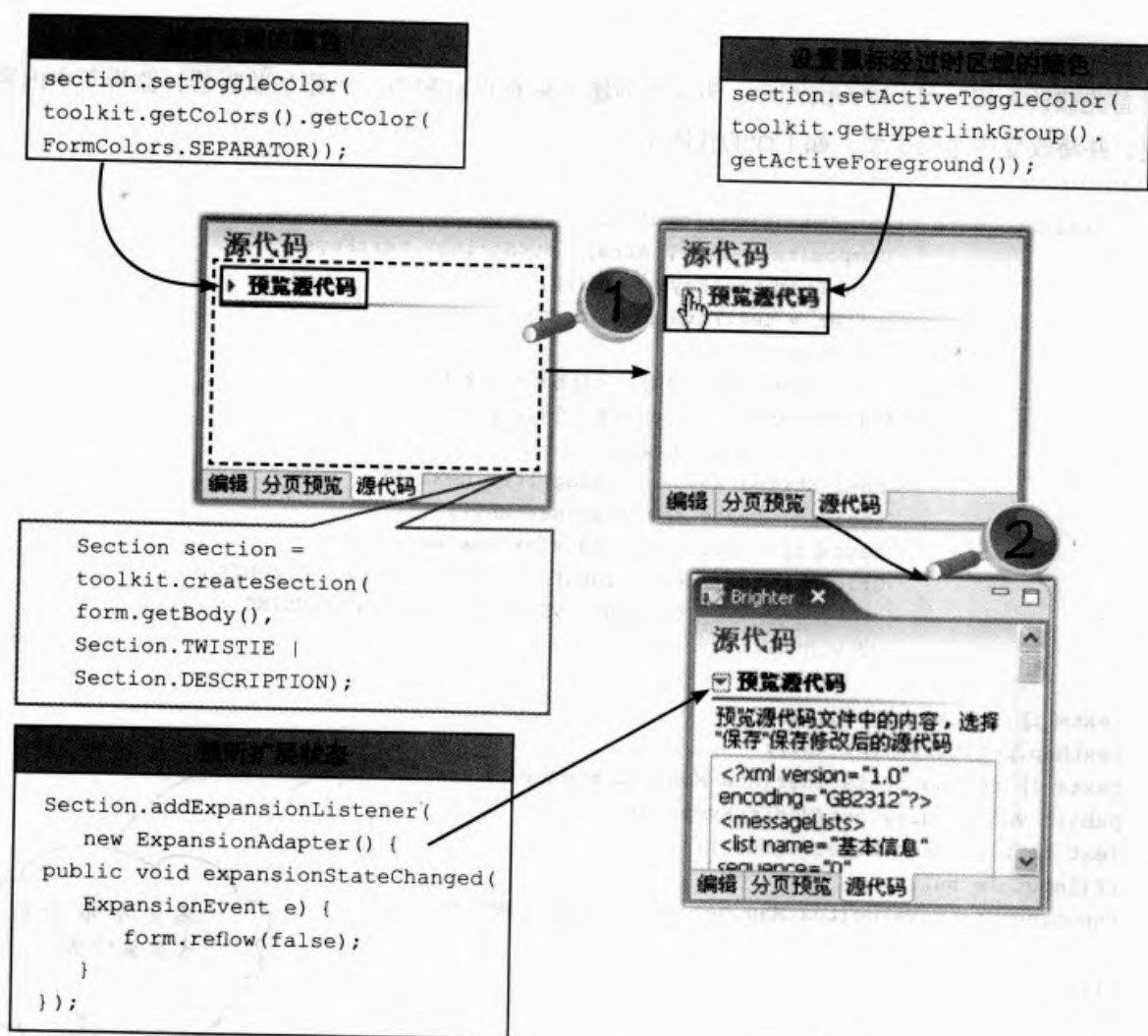


图13-26 “源代码”页的创建和监听过程

13.6 响应编辑器更改

在“编辑”页中，用户可以修改变编辑器中的内容（文本和选择框），但是这些修改必须要反映到模型中，最终存储到计算机里。当应用程序重新打开时，所有的数据再从计算机中恢复到上一次关闭时的状态，这样的数据才是有意义的，因而，必须在编辑器做出更改后，同时修改相应的模型，以达到模型和显示的一致，最终保存用户数据的目的。

同时，还需要更新“分类预览”页面中的数据，保证其加载的是最新的模型。下面分别介绍实现的方法。

13.6.1 更新模型

AddressListProperties中定义了AddressList类型的全局变量input，该变量管理和维护相应的

AddressList生命周期中的数据。

首先修改AddressListProperties, 为每个创建出来的Text和choices添加监听器, 监听其中内容的改变, 并将改变传递给模型, 如下面的代码所示。

```
private void createTextItems(FormToolkit toolkit,
    Composite parent, ArrayList<String> textKeys){
    texts = new Text[ textKeys.size()];
    Iterator iter = textKeys.iterator();
    //遍历TextContentsItems
    //为每个TextContentsItem创建一个标签和一个文本框
    //并且为每个TextContentsItem添加一个修改监听器
    for(int i = 0; iter.hasNext(); i++){
        final String key = (String)iter.next();
        toolkit.createLabel(parent, key);
        texts[i] = toolkit.createText(parent, "");
        GridData gd = new GridData(GridData.FILL_HORIZONTAL|
            GridData.VERTICAL_ALIGN_BEGINNING);
        gd.widthHint = 10;

        texts[i].setLayoutData(gd);
        textMap.put(texts[i], key);
        texts[i].addModifyListener(new ModifyListener(){
            public void modifyText(ModifyEvent e){
                Text text = (Text)e.getSource();
                if(input != null)
                    input.setStringValue(textMap.get(text), text.getText());
            }
        });

        createSpacer(toolkit, parent, 2);
    }

    private void createChoicesItems(FormToolkit toolkit,
        Composite parent, Map<String, Object[]> choiceKeysMap){
        //有keys.size()个选择项
        Iterator iter = choiceKeysMap.keySet().iterator();
        choices = new Button[ choiceKeysMap.size()][];
        //遍历choiceContentsItems, 为每个choiceItem创建一个标签和多个选项按钮
        //并且为每个choiceContentsItem添加一个选择监听器
        for(int i = 0; iter.hasNext(); i++){
            GridData gd;

            SelectionListener choiceListener = new SelectionAdapter() {
                public void widgetSelected(SelectionEvent e) {
                    Button button = (Button)e.getSource();
                    Integer value = (Integer)e.widget.getData();
                    if (input!=null) {
                        input.setIntValue(buttonMap.get(button), value);
                    }
                }
            };
            //创建选择适配器
```

为文本框添加
修改监听器

创建选择适配器

```

};
String key = (String)iter.next();
//在此处创建Label
Label label = toolkit.createLabel(parent, key + ": ");
GridData gdata = new GridData(GridData.FILL_HORIZONTAL);
gdata.horizontalSpan = 2;
label.setLayoutData(gdata);
Object[] choiceStrings = (Object[])choiceKeysMap.get(key);
choices[i] = new Button[choiceStrings.length];
//为选择按钮添加选择监听器
//在同一个choiceContentItem中的一组选择按钮对应相同的选择监听器
for(int j = 0; j < choices[i].length; j++){
    choices[i][j] = toolkit.createButton(parent,
        (String)choiceStrings[j], SWT.RADIO);
    choices[i][j].setData(new Integer(j));
    buttonMap.put(choices[i][j], key);
    choices[i][j].addSelectionListener(choiceListener);
}
gd = new GridData();
gd.horizontalSpan = 1;
choices[i][j].setLayoutData(gd);
}
createSpacer(toolkit, parent, 2);
}
}

```

为每个按钮添加选择适配器



在createTextItems方法中的修改监听器和createChoicesItems方法中的选择适配器中都引用了AddressList的setter方法，这类方法将修改相应模型中的值，并且通知对模型值更改感兴趣的对象（参见第13.4.2节“数据管理模型”）。

当在Master部分重新选择编辑过的AddressList模型后，selectionChanged()方法将会执行，它会调用update()方法来重新读取模型中的数据，更新相应的显示。其代码如下所示。

```

public void selectionChanged(IFormPart part, ISelection selection) {
    IStructuredSelection ssel = (IStructuredSelection)selection;
    if (ssel.size()==1) {
        input = (AddressList)ssel.getFirstElement();
    }
    else
        input = null;
    update();
}

protected void update() {
    for(int i = 0; i < choices.length; i++)
        for(int j = 0; j < choices[i].length; j++){
            choices[i][j].setSelection(input!= null &&
            input.getIntValue(buttonMap.get(choices[i][j]))== j);
        }
    }

```

更新选择项


```
for(int i = 0; i < texts.length; i++)
{
    String key = textMap.get(texts[i]);
    texts[i].setText(input!=null &&
        input.getStringValue(key) != null?input.getStringValue(key): "");
}
```

更新文本区域

以BasicAddressList的更改和显示为例，显示的过程如图13-27所示。

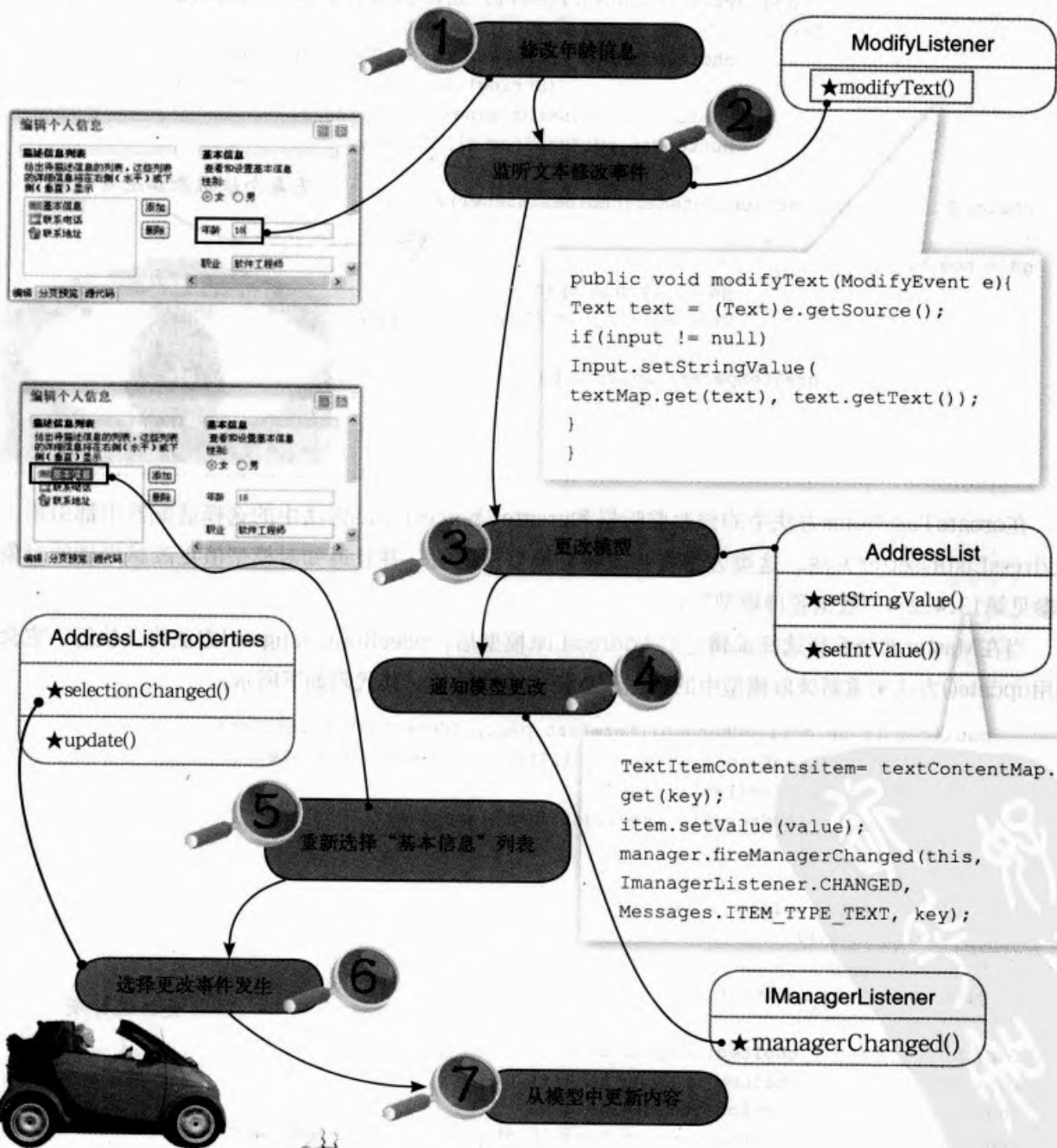
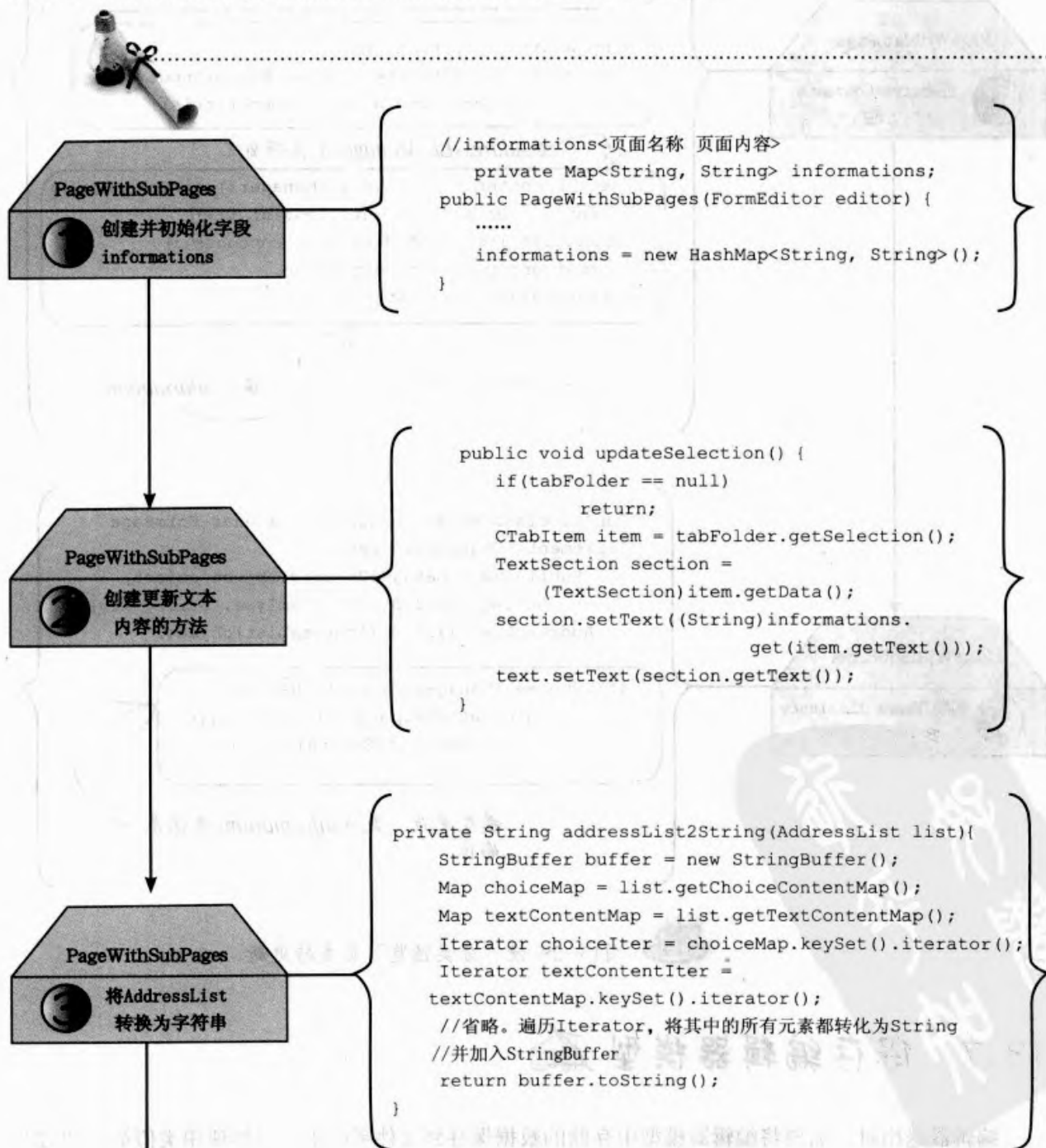


图13-27 编辑器修改和监听的过程

在图13-27中, 前 ① ~ ③ 步显示了用户更改“年龄”文本, 并修改模型的过程。当用户重新选择“基本信息”列表后(这中间可能进行若干操作), 将重新从修改后的模型中读入内容, 如图13-27中的步骤 ④ ~ ⑤ 所示。

13.6.2 使“分页预览”页支持更新

“分页预览”页需要分类显示模型的内容, 并且在模型更改时刷新文本框的显示。为此, 需要按如图13-28所示的步骤更改PageWithSubPages类, 使其支持内容更新。



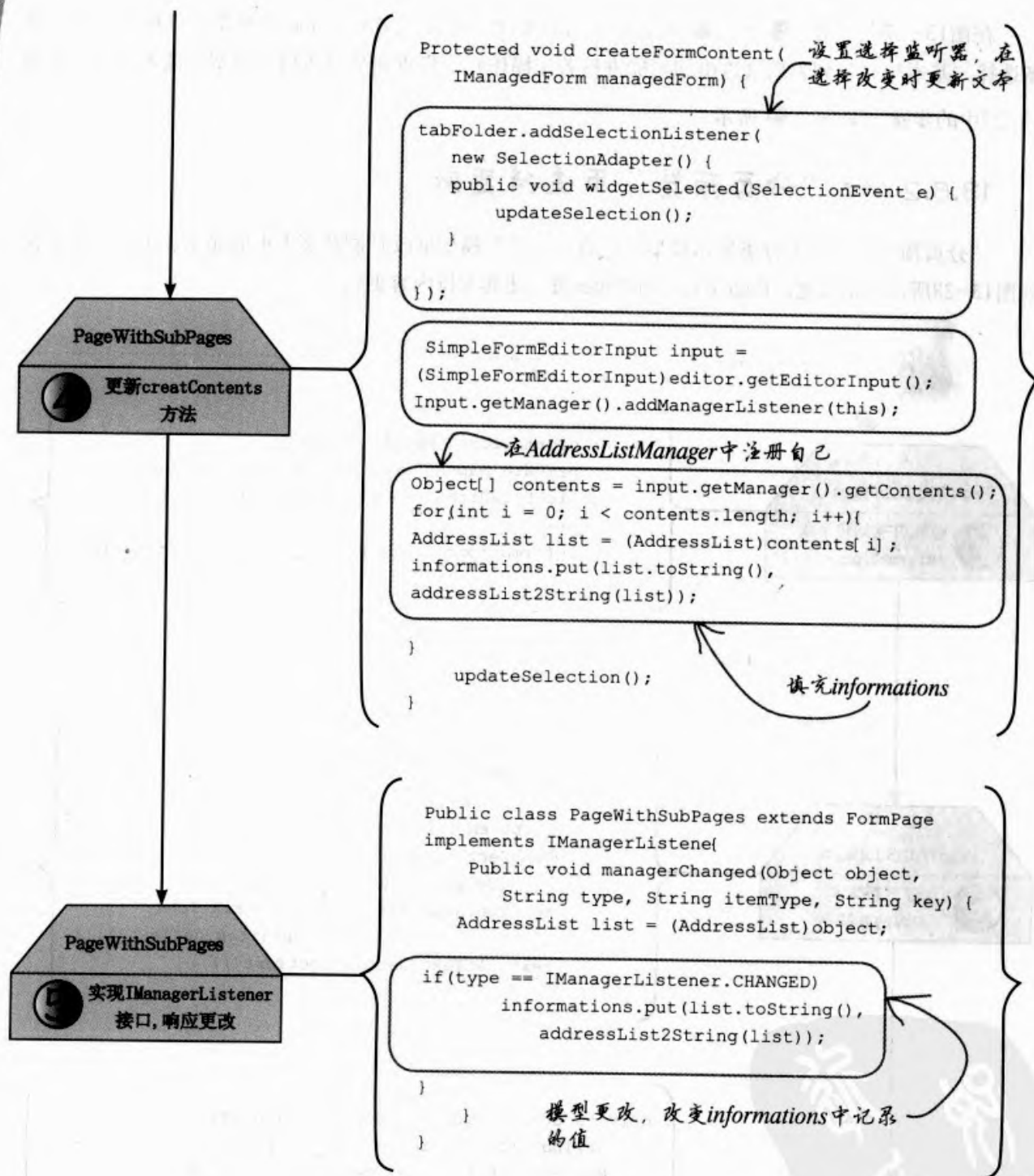


图13-28 使“分类预览”页支持更新

13.7 保存编辑器模型

编辑器退出时，需要将编辑器模型中存储的数据保存到文件系统中，仍然使用类似第12.8.2节

“存储视图元素信息”时所用的方法。但是，由于XMLMemento中仅能使用以UTF-8编码的字符，需要修改类MYXMLMemento（见第13.4.3节“文件映像模型”）来更改默认的字符。更改该类中的私有类DOMWriter中的XML_VERSION的值，以使该DOMWriter支持中文，代码如下所示。

```
private static final String XML_VERSION =
    "<?xml version=\"1.0\" encoding=\"GB2312\"?>";
```

然后，设计同地址本中的编辑信息相对应的XML文件的格式，如图13-29所示。

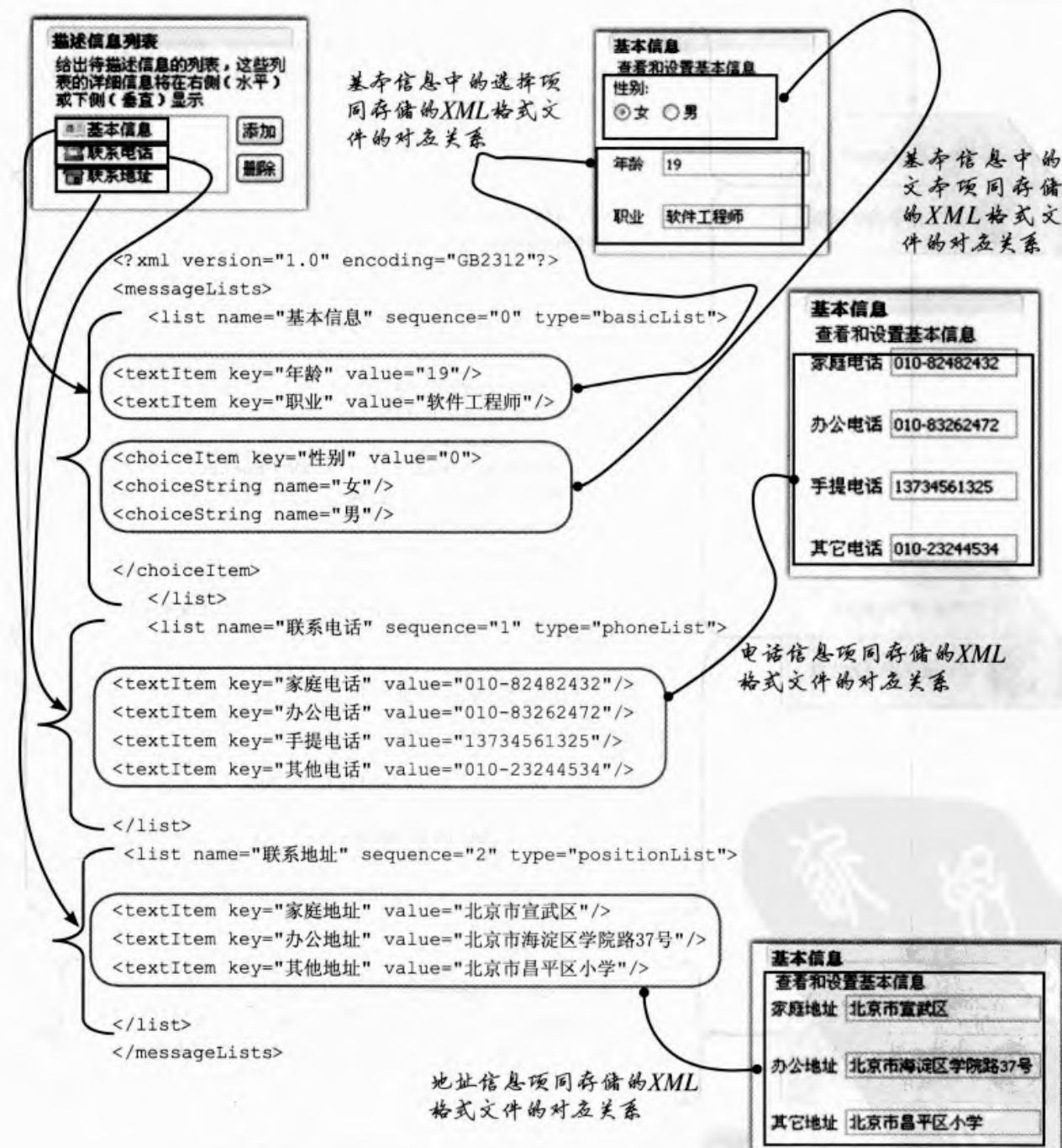
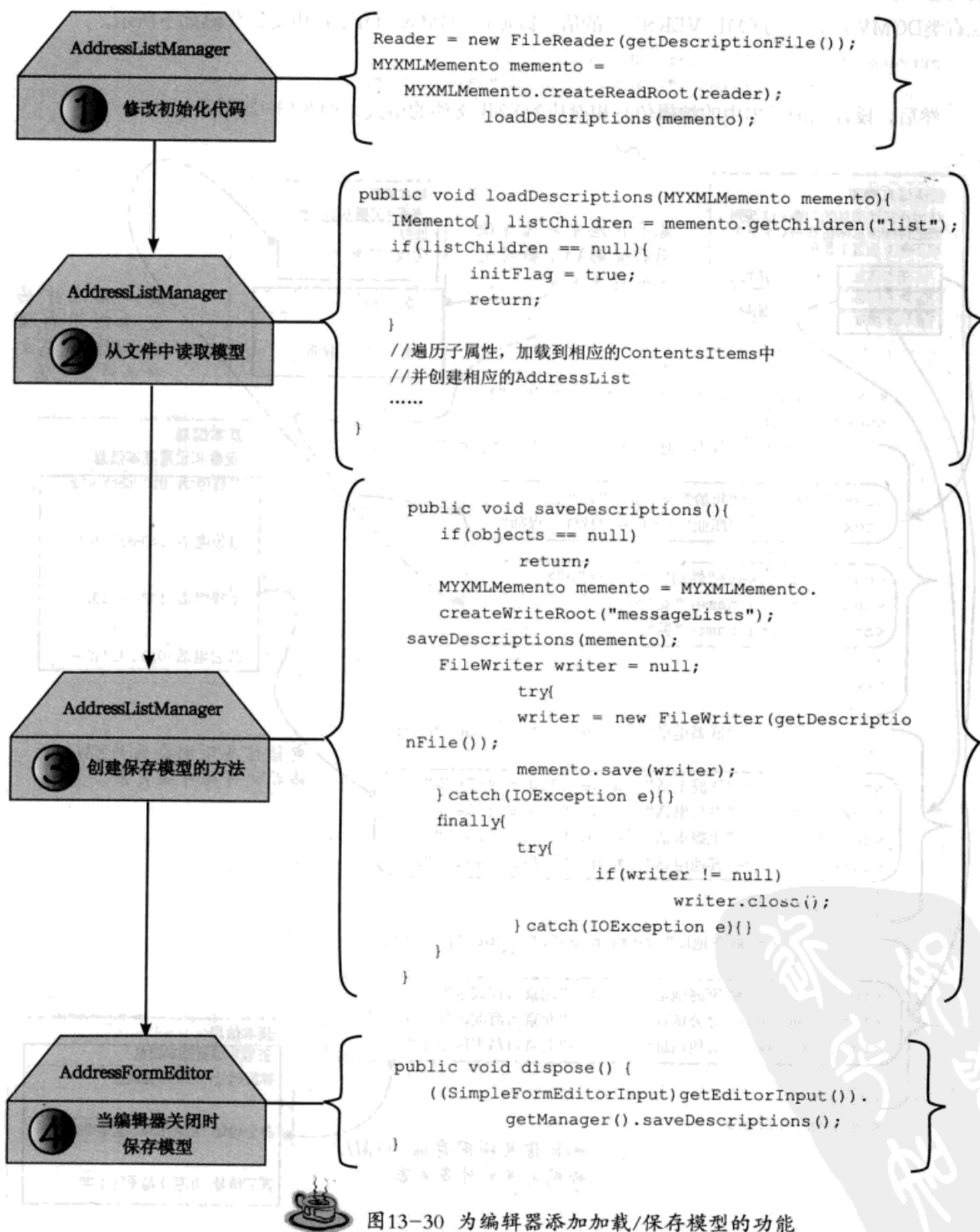


图13-29 保存的XML文件同界面UI元素的对应关系

设计好编辑器模型后, 参照如图13-30所示的步骤来保存编辑器模型。在图13-30中仅列出了部分代码, 查看完整代码请参阅随书光盘。



完成上述步骤后, 就可以保存文件了, 文件保存在工作空间元数据的如下子目录中。

<workspaceDir>\.metadata\plugins\com.plugindev.addressbook

※ 注意 ※

图13-30中的第4步将保存模型方法的调用放在了`dispose()`中,这意味着在用户关闭编辑器时会调用`AddressManager`中的`saveDescription()`方法。但是,细心的读者很容易就会发现,这同第12章介绍的视图的保存没有任何的区别。别急,这只是本节所做的替代方法,在下一节编辑器生命周期中将会对这种实现进行改进,使用编辑器的`doSave()`方法来保存此模型。



接下来,修改“源代码”页,为其添加`loadSource()`方法,使其支持源代码的显示(在编辑器的生命周期过程中,该页中的代码不发生变化),支持源代码浏览的代码如下所示。

```
public void loadSource() {
    if(text == null)
        return;

    File file = Activator.getDefault().getStateLocation().
        append(peopleName + ".xml").toFile();

    if(file.exists() == false){
        text.setText("");
        return;
    }

    StringBuffer buffer = new StringBuffer();
    try {
```

根据人名获得相应的.xml文件

```
        BufferedReader reader;
        reader = new BufferedReader(
            new InputStreamReader(new FileInputStream(file)));
```

创建BufferedReader

```
        String temp = reader.readLine();
```

```
        while(temp != null)
        {
            if(temp.contains("<list") ||
                temp.contains("</list>"))
                buffer.append("\t");
            else if(temp.contains("<textItem"))
                buffer.append("\t\t");
            else if(temp.contains("<choiceItem") ||
                temp.contains("</choiceItem>"))
                buffer.append("\t\t");
            else if(temp.contains("<choiceString"))
                buffer.append("\t\t\t");
            buffer.append(temp);
            buffer.append("\n");
            temp = reader.readLine();
        }
```

根据XML元素所处的不同层级,
为其插入制表符

```
    } catch (FileNotFoundException e) {
```




```

        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally{

```

将读取到的字符串放入文本框中

```

        text.setText(buffer.toString());
    }
}

```

13.8 编辑器生命周期

通常的编辑器要经过“打开—修改—保存—关闭”的生命周期，地址本编辑器也不例外。当编辑器打开时，调用init(IEditorSite, IEditorInput)方法来设置编辑器的初始内容。在用户修改编辑器的内容后，编辑器需要通知注册的监听器它的内容已经处于“已修改”状态。当用户保存编辑器内容之后，还需要通知注册的监听器该编辑器的内容不再处于“已修改”状态。

对编辑器的监听由Eclipse自动完成，这些监听器基于isDirty()方法的返回值执行各种任务，如更新编辑器的标题、增加或去掉编辑器标题前面的星号以及更改“保存”操作的状态等。当关闭编辑器时，如果isDirty()方法返回值为真，Eclipse会自动弹出对话框询问是否保存编辑器的内容。

13.8.1 已修改的编辑器

编程人员需要确保编辑器知道自从上次保存操作之后，用户是否修改了编辑器的内容。如下示例程序通过使编辑器实现IManagerListener的方法，并且引入字段isPageModified来达到这个目的。

```

public class AddressFormEditor extends FormEditor implements IManagerListener{
    private boolean isPageModified;
    .....
    public void managerChanged(Object object, String type, String itemType, String key) {
        if(isPageModified == false){
            isPageModified = true;

```

```

        editorDirtyStateChanged();
    }
}

```

更改编辑器状态，变“已保存”状态为“脏”状态

在上面的代码中，editorDirtyStateChanged()方法更改了编辑器的状态，并通知监听器执行相应的操作。editorDirtyStateChanged()方法也可以变“脏”（已修改）状态为“已保存”状态，见第13.8.3节“保存内容”。

为了实现编辑器对模型内容的监听，还需要在addPages()方法中为AddressListManager注册自身，代码如下所示。

```
((SimpleFormEditorInput) getEditorInput()).
    getManager().addManagerListener(this);
```

最后，需要让其他注册的监听器知道编辑器的内容处于已修改状态，这是通过设置isDirty()方法获得，代码如下所示。

```
public boolean isDirty(){
    return isPageModified || super.isDirty();
}
```

用来提供开发者自己设定的编辑器已修改的指示

现在，只要修改了编辑器中的内容，都需要设置新的isPageModified字段。但是，IManagerListener会监听一切更改模型的操作，这包括初始化时从文件中加载模型和重新读取AddressList模型时（只要设置Details部分的文本或选择按钮的值，模型更改事件就会发生），而这并不符合开发的理念。为此还需要更改AddressListProperties类中的相应代码，为其加入字段updated，并更改其中的update()方法，代码如下所示。

```
protected void update() {
    if(updated == false)
        updated = true;
    //从模型中更新显示
    .....
    updated = false;
}
```

当调用此方法时updated初始值为false，表明不为初始化或重新读取模型时的更新

修改完毕，updated设为false



然后，再设置相应的监听器，使update为真时，不更新模型的值，因为此时文本中的内容是从模型中读入的，同模型中的内容相同。以文本项为例，代码如下所示。

```
texts[i].addModifyListener(new ModifyListener(){
    public void modifyText(ModifyEvent e){
        if(updated == true)
            return;
        Text text = (Text)e.getSource();
        if(input != null)
            input.setStringValue(textMap.
                get(text), text.getText());
    }
});
```

当此时updated初始值为true，表明为初始化或重新读取模型时的更新，因而不调用update()方法




13.8.2 切换页面

当在“编辑”页和“分类预览”页之间切换时，所有在“操作”页面所做的修改必须反映到“分类预览”页中，为了达到此目的，可以覆盖pageChange(int)方法，在用户选择“分类”预览页时，更新页面内容，代码如下所示。

```
public void pageChange(int newPageIndex){
    switch(newPageIndex){
        case 0:
            break;
        case 1:
            if(isPageModified)
            {
                PageWithSubPages page = (PageWithSubPages)getSelectedPage();
                page.updateSelection();
            }
            break;
    }
    super.pageChange(newPageIndex);
}
```

如果页面有改动，则更新“分类预览”页中的显示



13.8.3 保存内容

第13.7节曾经提到，使用dispose()方法保存编辑器只有当编辑器关闭时才能保存，而且用户并不清楚什么时候编辑器发生更改。有时候用户修改编辑器内容后，并不希望保存新的信息，这时就不能使用dispose()方法来保存编辑器了，需要启用doSave()方法，其实现如下所示。

```
public void doSave(IProgressMonitor monitor) {
    if(isPageModified){
        ((SimpleFormEditorInput)getEditorInput()).
            getManager().saveDescriptions();
        sourceChanged = true;
    }
    isPageModified = false;

    if(this.getActivePage() == 2)
    {
        SourcePage page= (SourcePage)getSelectedPage();
        page.loadSource();
        sourceChanged = false;
    }

    editorDirtyStateChanged();

    updateTitle();
}
```

如果当前页面为“源代码”页面，则重新读入源代码

编辑器中的信息已保存，因而更改编辑器的“脏”状态

在上面这段代码中, editorDirtyStateChanged()通知监听器编辑器的属性已经由“脏”状态变为“已保存”状态,从而触发监听器做出相应的动作。

保存编辑器内容之后,还需要更新“源代码”页面文本框中的显示内容。这可以根据保存时当前页面的不同分为两种情况,即当前页面为“源代码”页和当前页面不为“源代码”页。

当前页面为“源代码”页面时,在doSave()方法中重新读入源代码(如上面的代码所示),如果当前页面不为“源代码”页面,可以使用pageChanged()方法,在切换到“源代码”页时更新文本框的显示,代码如下所示。

```
public void pageChange(int newPageIndex){
    .....
    case 2:
        SourcePage page = (SourcePage)getSelectedPage();
        if(sourceChanged == true){
            page.loadSource();
            sourceChanged = false;
        }
        break;
    .....
}
```

从源文件中获取内容

为编辑器加入上面的代码后,“保存”工具栏自动启用,同时编辑器的标题也会根据编辑器的状态在前面加入或去掉“*”,如图13-31所示。

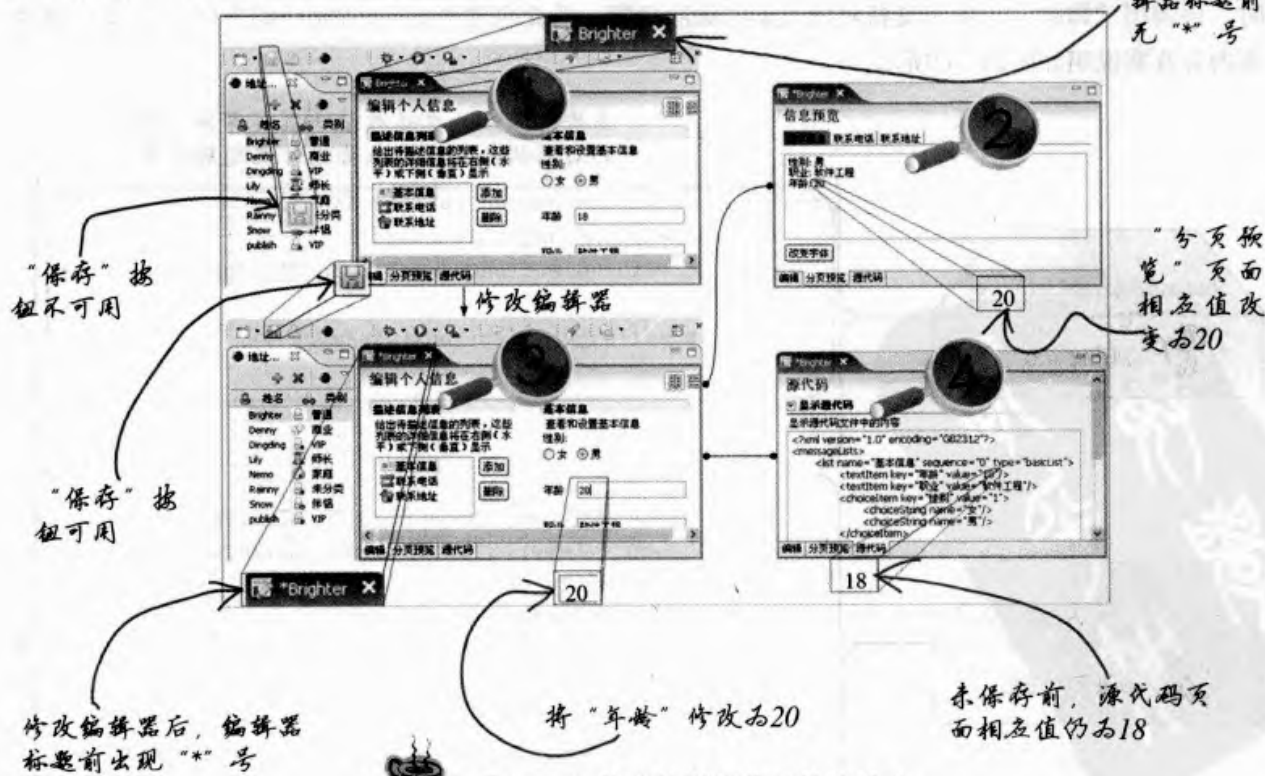


图13-31 修改编辑器前后的编辑器

从图13-31中可以看到,“编辑”页面修改后,“分页预览”页会根据“编辑”页面的改动更新显示的内容,而“源代码”页却没有改动,这是因为修改的内容还没有保存到文件系统中;只有当“保存”命令执行后,“源代码”页面的内容才会更新。

13.9 为编辑器添加操作

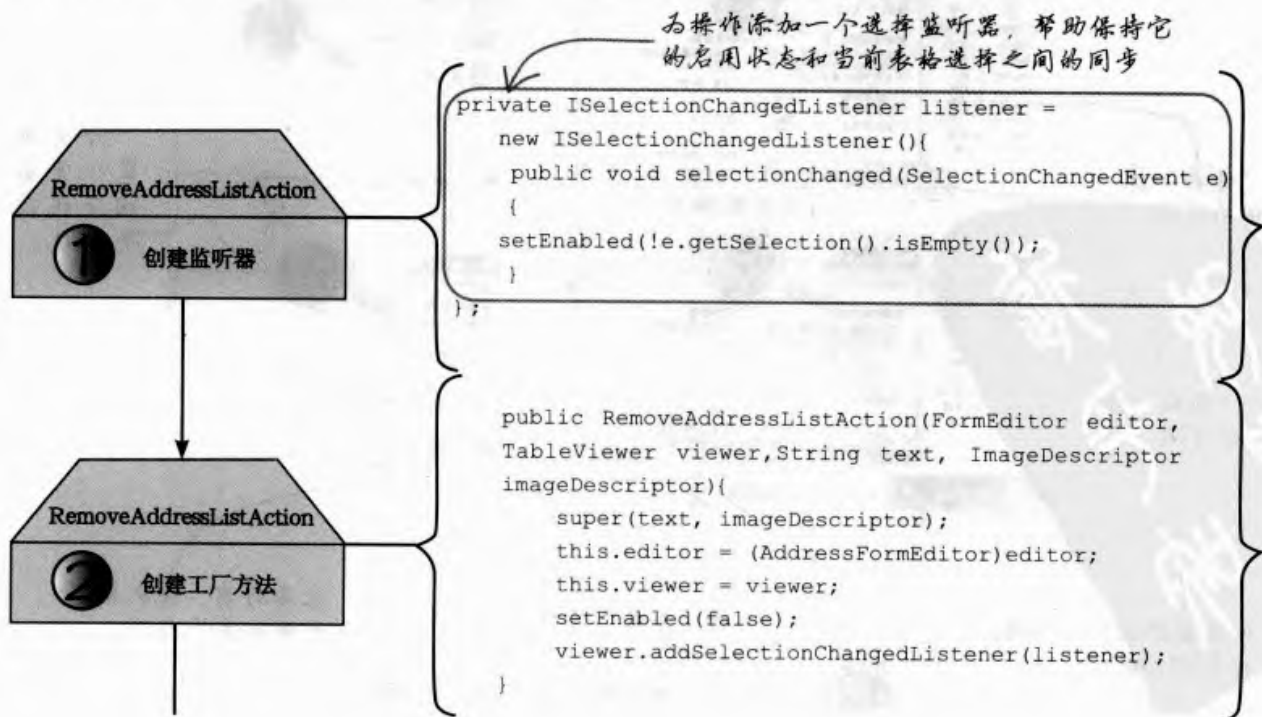
编辑器操作可以作为编辑器上下文菜单的菜单项出现,也可以作为工作台工具栏中的工具栏按钮出现,还可以作为工作台菜单中的菜单项出现。本节将介绍以编程的方式将操作添加到编辑器,而在第11.6节“编辑器操作”介绍的是如何使用插件清单中的声明来添加操作。

13.9.1 上下文菜单

编辑器一般具有用某些动态填充的上下文菜单,这些操作的对象是编辑器或在编辑器内选择的对象。创建编辑器的上下文菜单需要经历几个步骤,同时还需要另外几个步骤注册此编辑器,让其他编辑器可以添加操作(参见第11.6节“编辑器操作”,了解如何通过插件清单为编辑器添加上下文菜单)。本节通过为“编辑”页面添加“删除”操作来介绍如何为编辑器构建上文菜单。

1. 创建操作类

编辑页面的Master部分需要一个删除表格元素的操作,当用户选择AddressList元素并单击右键时,会弹出“删除”菜单来支持对该元素的删除功能。为此创建RemoveAddressListAction类,其主要内容及其说明如图13-32所示。



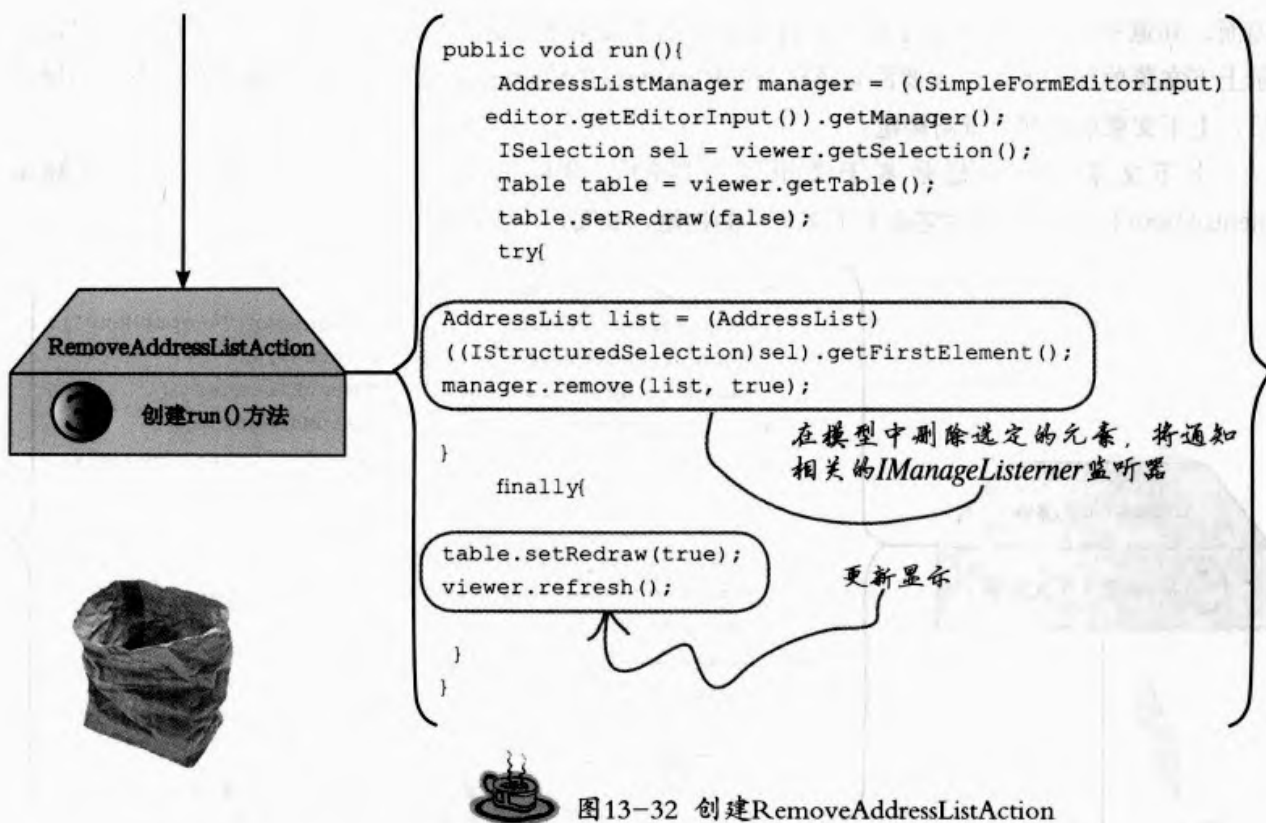


图13-32 创建RemoveAddressListAction

2. 创建操作

`RemoveAddressListAction`类创建好后，在`ScrolledPropertiesBlock`类中创建一个新字段来保存该操作，然后通过`createMasterPart()`方法来调用下面代码所示的新方法来初始化该字段。

```
private RemoveAddressListAction removeAction;
private void createActions(){
    ImageDescriptor removeImage =
        PlatformUI.
        getWorkbench().getSharedImages().
        getImageDescriptor(ISharedImages.IMG_TOOL_DELETE);
    ImageDescriptor disableRemoveImage =
        PlatformUI.
        getWorkbench().getSharedImages().
        getImageDescriptor(ISharedImages.IMG_TOOL_DELETE_DISABLED);
    removeAction =
        new RemoveAddressListAction(
            page.getEditor(),viewer,"删除", removeImage);
    removeAction.setDisabledImageDescriptor(disableRemoveImage);
}
```

在后面，同样的处理方式还将用作基于键盘的操作。

3. 动态构建上下文菜单

上下文菜单需要同编辑器页面一同创建。然而，由于插件需要基于不同的选择对象构建菜单项，

因而，知道用户在哪里单击了鼠标右键之后，上下文菜单项的内容才能够确定。为了能够正确地显示上下文菜单项的内容，需要设置菜单项的RemoveAllWhenShown为真，以便每次单击鼠标右键之后，上下文菜单项都会重新构建。

上下文菜单的构建是基于菜单监听器的，当单击鼠标右键之后，菜单监听器触发menuAboutToShow()方法完成上下文菜单的构建。如图13-33所示。

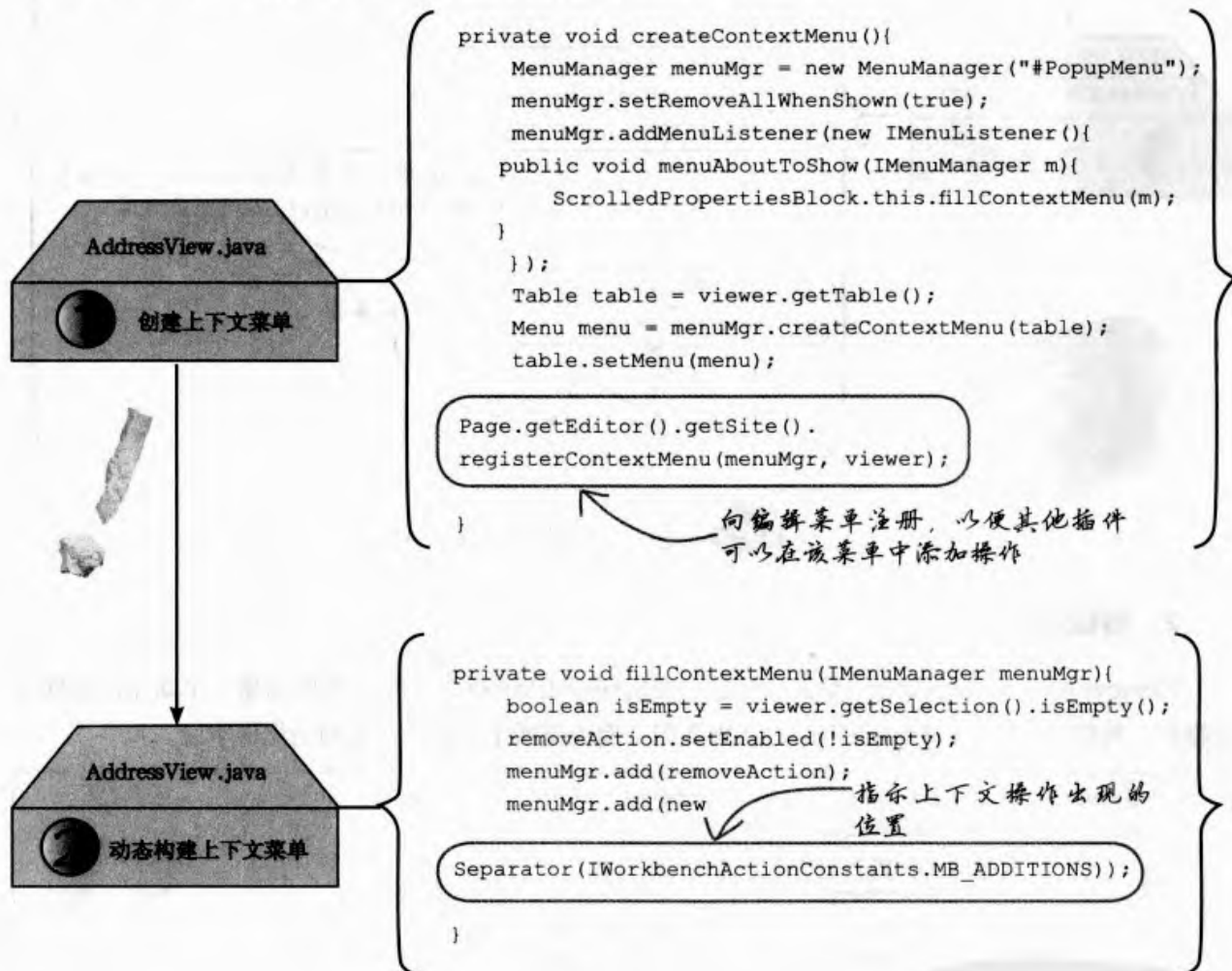


图13-33 构建上下文菜单的步骤

图13-33步骤●为上下文菜单添加了一个分隔符和一个IWorkbenchActionConstants.MB_ADDITIONS常量，指示添加的操作出现在上下文菜单的哪个地方。

完成所有步骤后，上下文菜单就包含在了“编辑”页面中，但是，为了使编辑器AddressListEditor支持“删除”功能，还需要在删除操作执行后，更新“分页预览”页中的显示（删除一个AddressList后，“分页预览”页面将减少一个子页面）。因此在PageWithSubPages类的managerChanged()方法中添加对模型删除的支持，代码如下所示。

```

public void managerChanged(Object object,
    String type, String itemType, String key) {
    AddressList list = (AddressList)object;
    
```

```

if(type == IManagerListener.REMOVED)
{
    tabFolder.setRedraw(false);
    CTabItem[] items = tabFolder.getItems();
    for(int j = 0; j < items.length;j++)

```

```

if(items[j].getText().equals(list.toString()))
items[j].dispose();

```

```

informations.remove(list.toString());
tabFolder.setRedraw(true);
}

```

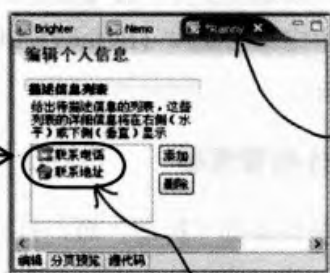
删除同指定元素相对应的CTabItem(子页面)

删除操作添加完成的效果如图13-34所示。

删除操作出现在上下文菜单中



删除



编辑器处于“修改”状态



编辑器已保存

保存后



“基本信息”列表项被删除

只剩两个子页面



图13-34 支持了“删除”操作后的编辑器

另外，“编辑”页的Master部分还创建了“删除”按钮，现在将其关联到删除操作上，使“删除”按钮可用（“添加”按钮的动作将在第15章“对话框和向导”中完成）。如下面的代码所示。

```

b.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
        removeAction.run();
    }
});

```

13.9.2 管理编辑器操作栏

接口org.eclipse.ui.IEditorActionBarContributor（操作栏管理器）的实例为一个或多个编辑器管理全局菜单、菜单项和工具栏按钮的安装和删除。同编辑器器匹配的操作栏管理器在插件清单中指明，它通常是org.eclipse.ui.part.EditorActionBarContributor的子类。在定义了操作栏管理器后，平台给其发送事件，指示编辑器何时变为活动，何时为不活动，从而使得编辑器操作栏管理器能够在

合适的时候安装或删除菜单和按钮。

IEditorActionBarContributor中声明的主要方法如下所示

★dispose()——当不再需要操作栏管理器时，自动调用该方法。它将清理该类创建的任何平台资源，如图像、剪切板等。

★init(IActionBars, IWorkbenchPage)——当第一次创建编辑器操作栏管理器时，调用这个方法。

★setActiveEditor(IEditorPart)——当关联的编辑器活动状态发生改变时，此方法被执行。操作栏管理器根据编辑器的活动状态来适当地安装或删除菜单和按钮。

EditorActionBarContributor类在实现IEditorActionBarContributor接口外，还提供了如下两个getter方法。

★getActionBars()——返回初始化操作栏管理器时，提供给管理器的操作栏。

★getPage()——返回初始化操作栏管理器时，提供给管理器的工作台页面。

本节通过将编辑器中定义的操作 (RemoveAddressListAction) 添加到顶层菜单和工具栏来介绍操作栏管理器的使用方法。

1. 创建编辑器操作栏管理器

首先，在插件清单编辑器的“扩展”页，打开编辑器扩展点，在扩展“地址本编辑器”的“属性”部分设置contributorClass为com.plugindev.addressbook.editors.AddressFormEditorContributor，然后，创建AddressFormEditorContributor类，创建的代码如下所示。

```
public class AddressFormEditorContributor extends
    EditorActionBarContributor {
    private static final String[] WORKBENCH_ACTION_IDS =
        { ActionFactory.DELETE.getId() };
    public void setActiveEditor(IEditorPart part){
        AddressFormEditor editor = (AddressFormEditor)part;
        setActivePage(editor, editor.getActivePage());
    }
    public void setActivePage(AddressFormEditor editor, int pageIndex){
        IActionBars actionBars = getActionBars();
        if(actionBars != null){
            switch(pageIndex){
                case 0:
                    MasterDetailsPage page =
                        (MasterDetailsPage)editor.getSelectedPage();
                    if(page != null)
                        hookGlobalTableActions(page, actionBars);
                    break;
            }
            actionBars.updateActionBars();
        }
    }
    public void hookGlobalTableActions(MasterDetailsPage page,
        IActionBars actionBars){
        for(int i = 0; i < WORKBENCH_ACTION_IDS.length; i++){
            actionBars.setGlobalActionHandler(WORKBENCH_ACTION_IDS[i],
                page.getTableAction(WORKBENCH_ACTION_IDS[i]));
        }
    }
}
```

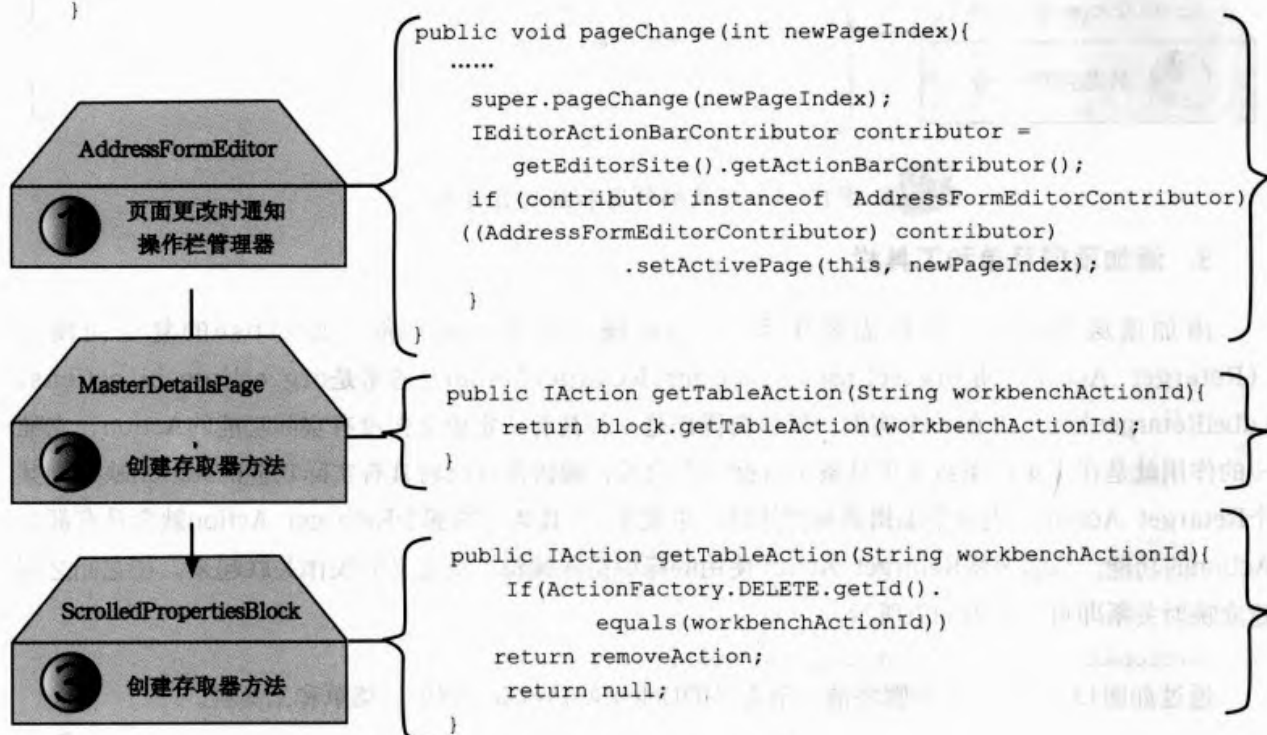
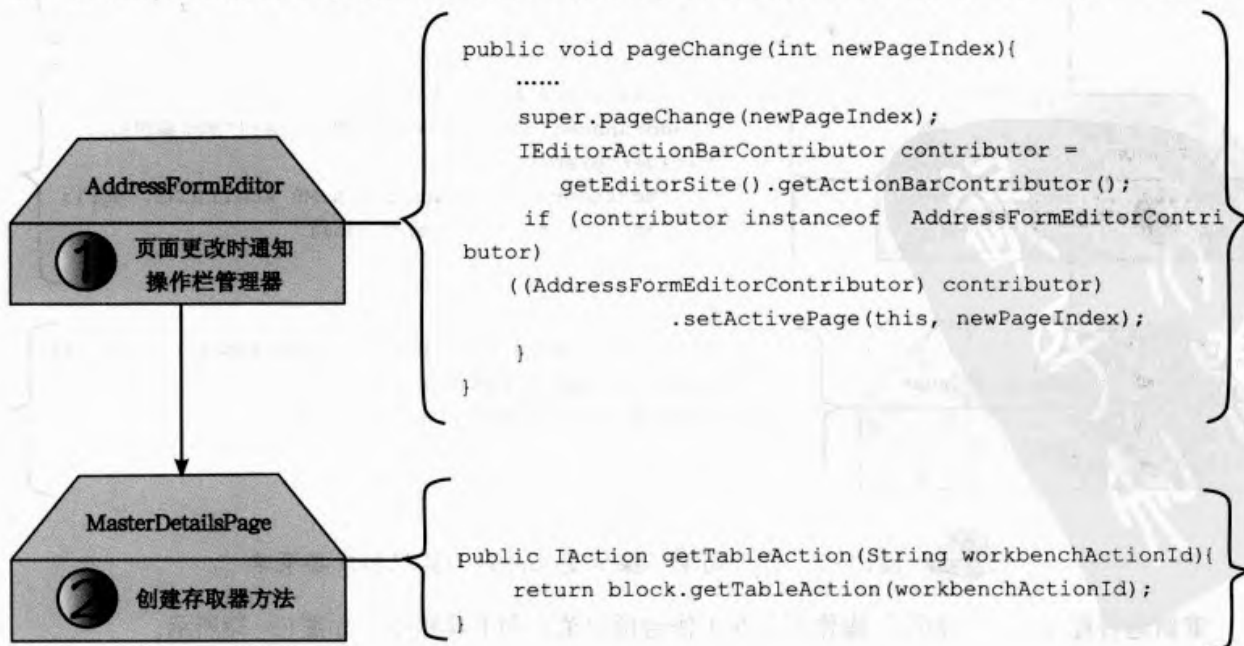


图13-35 修改编辑器和编辑器页面

2. 修改编辑器和编辑器页面

然后，修改地址本编辑器，在页面更改时通知操作栏管理器，以便操作栏管理器可以恰当地更新菜单项和工具栏按钮，同时修改编辑器页面，为操作栏管理器添加存取器方法。其步骤如图13-36所示。



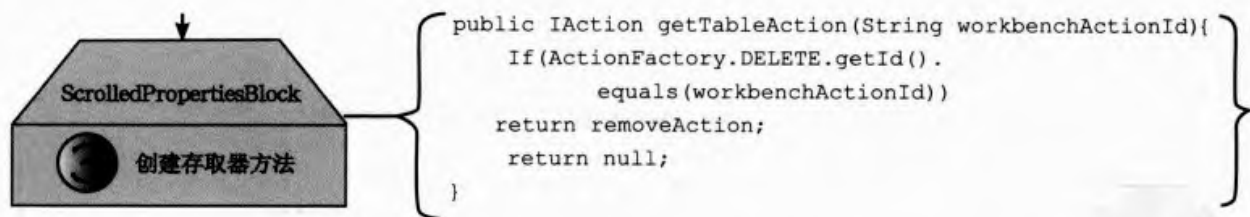


图13-36 修改编辑器和编辑器页面

3. 添加顶层菜单和工具栏

添加顶层菜单和工具栏需要使用Eclipse操作的复位向功能。Eclipse的复位向操作(Retarget Action)是org.eclipse.ui.actions.RetargetAction (通常是org.eclipse.ui.actions.LabelRetargetAction) 的一个实例。复位向操作是一种具有一定语义但没有实际功能的Action, 它唯一的作用就是在主菜单条或主工具条上占据一个位置, 编辑器可以将具有实际功能的Action映射到某个Retarget Action, 当这个编辑器被激活时, 主菜单/工具条上的那个Retarget Action就会具有那个Action的功能。只需要将Retarget Action使用的标识符同编辑器所定义的操作关联起来, 在它们之间建立映射关系即可, 代码如下所示。

```
setGlobalActionHandler(String, IAction);
```

通过如图13-37所示的步骤将前一节定义的DeleteAction添加到顶层菜单和工具栏。

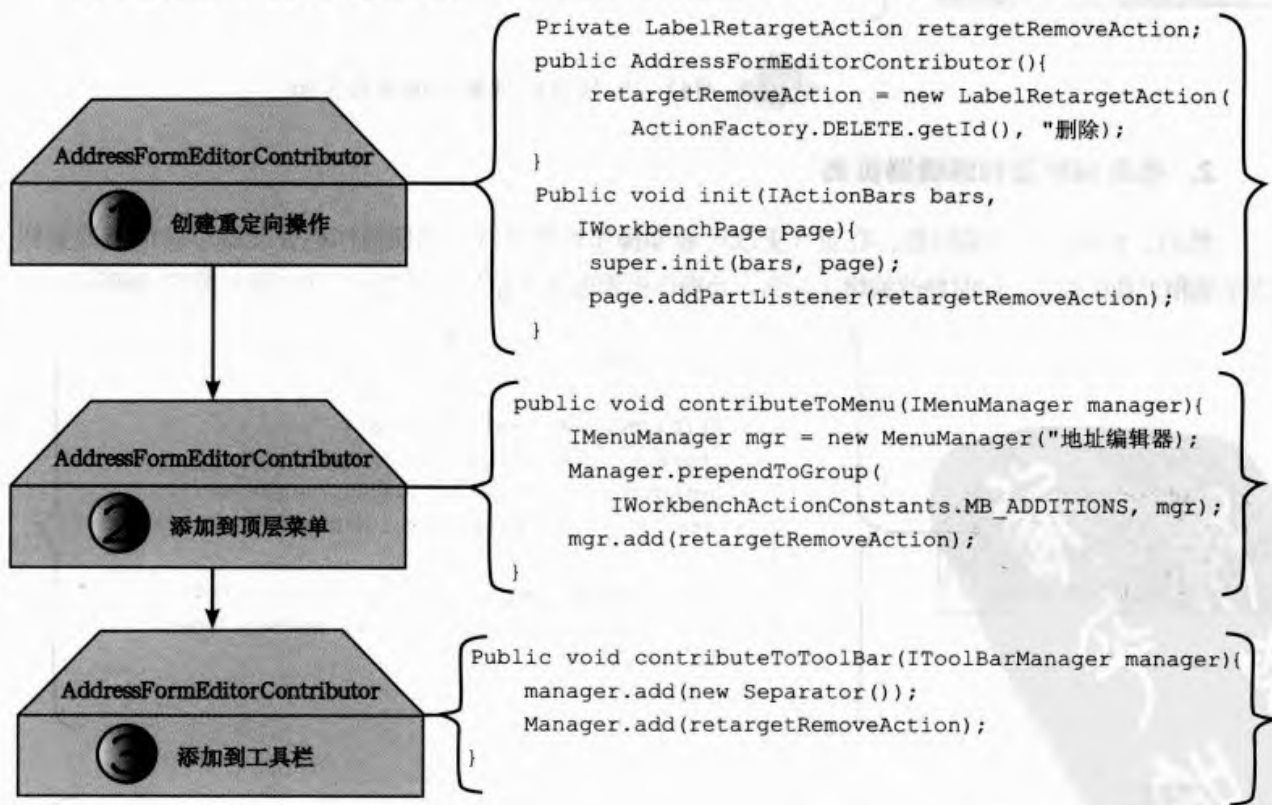


图13-37 将“删除”操作复位向到工具栏和顶层菜单

重新运行程序后, “删除”操作出现在工作台顶层菜单和工具栏中, 如图13-38所示。

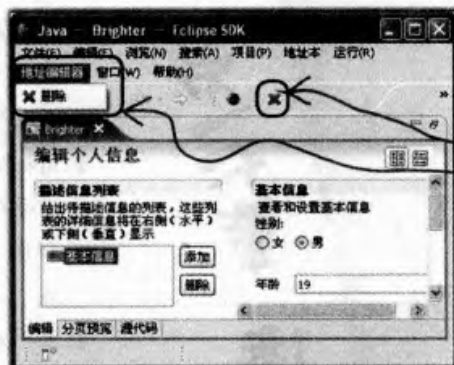


图13-38 将“删除”操作添加到工作台菜单和工具栏中的效果图

4. 添加键盘操作

要对“删除”操作添加键盘支持，需要在“删除”操作中添加键盘监听器，通过addTableKeyListener()来挂接删除键，是用户在按下该键时，删除在表格中选择的AddressList。

```
viewer.getTable().addKeyListener(new KeyListener(){
    public void keyPressed(KeyEvent e){
        if(e.character == SWT.DEL)
            removeAction.run();
    }
    public void keyReleased(KeyEvent e){
    }
});
```

13.10 本章小结

本章详细介绍了如何在工作台中创建新的编辑器；介绍了如何设置多页面表格编辑器、处理编辑器声明周期和创建编辑器的操作。

构建和定制编辑器是非常耗时的工作，Eclipse通过提供完善的编辑器架构，使得编辑器比较容易被构建。但是，虽然Eclipse提供了非常便捷的方式来搭建编辑器，但是要对编辑器的各个方面进行定制需要许多细化完善的操作，因而插件开发者应该专注于处理编辑器的特定行为，并尽可能地重用Eclipse现有的编辑器架构来处理通用的行为。

1. 回顾一下本章编辑器实现，构造一个特定的编辑器需要如下几个步骤。
2. 为特定的需求选择编辑器的类型；
3. 声明编辑器扩展；
4. 创建编辑器类；
5. 创建编辑器输入，实例化模型；
6. 提供打开编辑器的途径；
7. 定义编辑器的用户界面；
8. 处理编辑器输入的用户修改；
9. 同步模型和编辑器；
10. 处理编辑器输入的存储；
11. 定义编辑器操作。
12. 下一章将介绍Eclipse的透视图。



第14章

透视图 (Perspectives)

拉开崭新的学习帷幕

完成了视图和编辑器的构建之后，读者可能会这样想，如果希望创建的视图和编辑器在一起显示和消失（因为它们的功能是匹配的），这样的想法可能实现吗？Eclipse给出了肯定的答案，它的可扩展性允许开发者把几个需要的视图和编辑器组合起来，进而创建一个新定制的透视图。本章将介绍如何为地址本插件创建一个新的透视图，并且在该透视图中添加“地址本”编辑器和“地址本”视图。

本章内容包括：

- ★介绍透视图的概念。
- ★添加透视图扩展，创建透视图。
- ★介绍透视图的显示布局IPageLayout。
- ★介绍如何填充和扩展现有透视图。



进入第14章

14.1 什么是透视图

透视图是UI层的概念，它实际上是视图和操作的集合，也可以包含一个对特定的用户人物和角色有用的编辑器区域。一个工作台窗口可以包括多个透视图，如果把工作台窗口看做一本书的话，每个透视图就是书的一页，任何时候只能看到其中的一个页面，如图14-1所示。

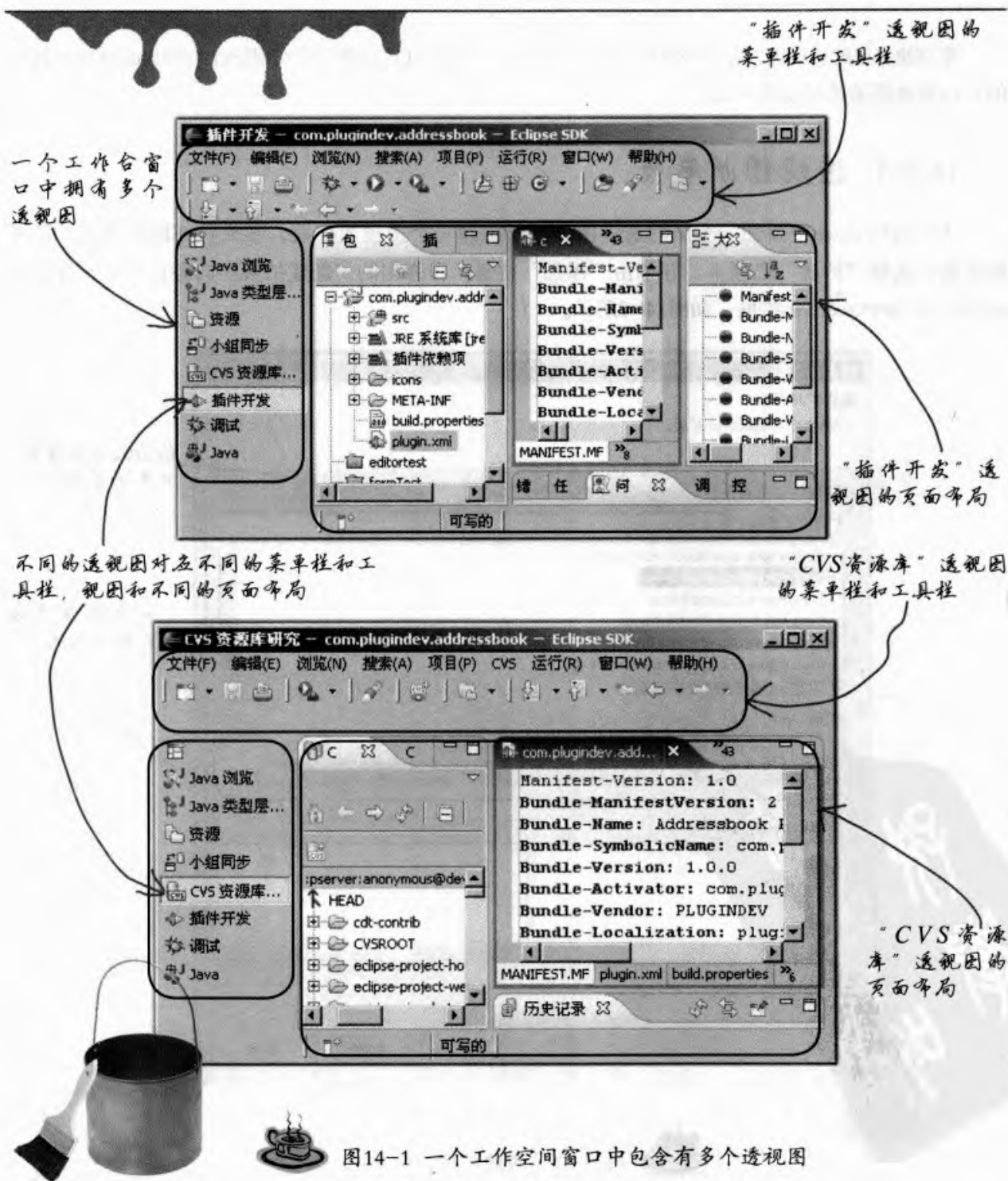


图14-1 一个工作空间窗口中包含有多个透视图

Eclipse的透视图为包含了多个编辑器和视图的大型插件和应用程序提供了组合的可能性, 插件开发者可以把提供的透视图拼接起来, 让用户可以更容易地了解应该如何组织产品的视图、编辑器和访问关键操作。

14.2 创建一个透视图

要创建自己的透视图, 首先需要扩展透视图扩展点, 然后通过创建一个实现IPerspectiveFactory接口的Java类来描述透视图的布局。

14.2.1 透视图扩展点

为了在Eclipse中添加一个新的透视图, 需要新建一个透视图扩展。首先打开地址本插件清单编辑器, 选择“扩展”选项卡, 并单击“添加”按钮, 在弹出的“新建扩展”向导页面中选择org.eclipse.ui.perspectives扩展, 如图14-2所示。



图14-2 “新建扩展”向导

单击“完成”按钮，将该扩展添加到插件扩展列表中。在插件清单编辑器的“扩展”页面中，右键单击加入的org.eclipse.ui.perspectives扩展点，在弹出的上下文菜单中选择“新建”→“Perspective”，在右侧的“扩展元素详细信息”部分填入如图14-3所示的属性。

由图14-3可以看到，org.eclipse.ui.perspectives所具有的属性如下所示。

★id★——用于引用透视图的唯一标识符。

★name★——和透视图相关联的文本标签。

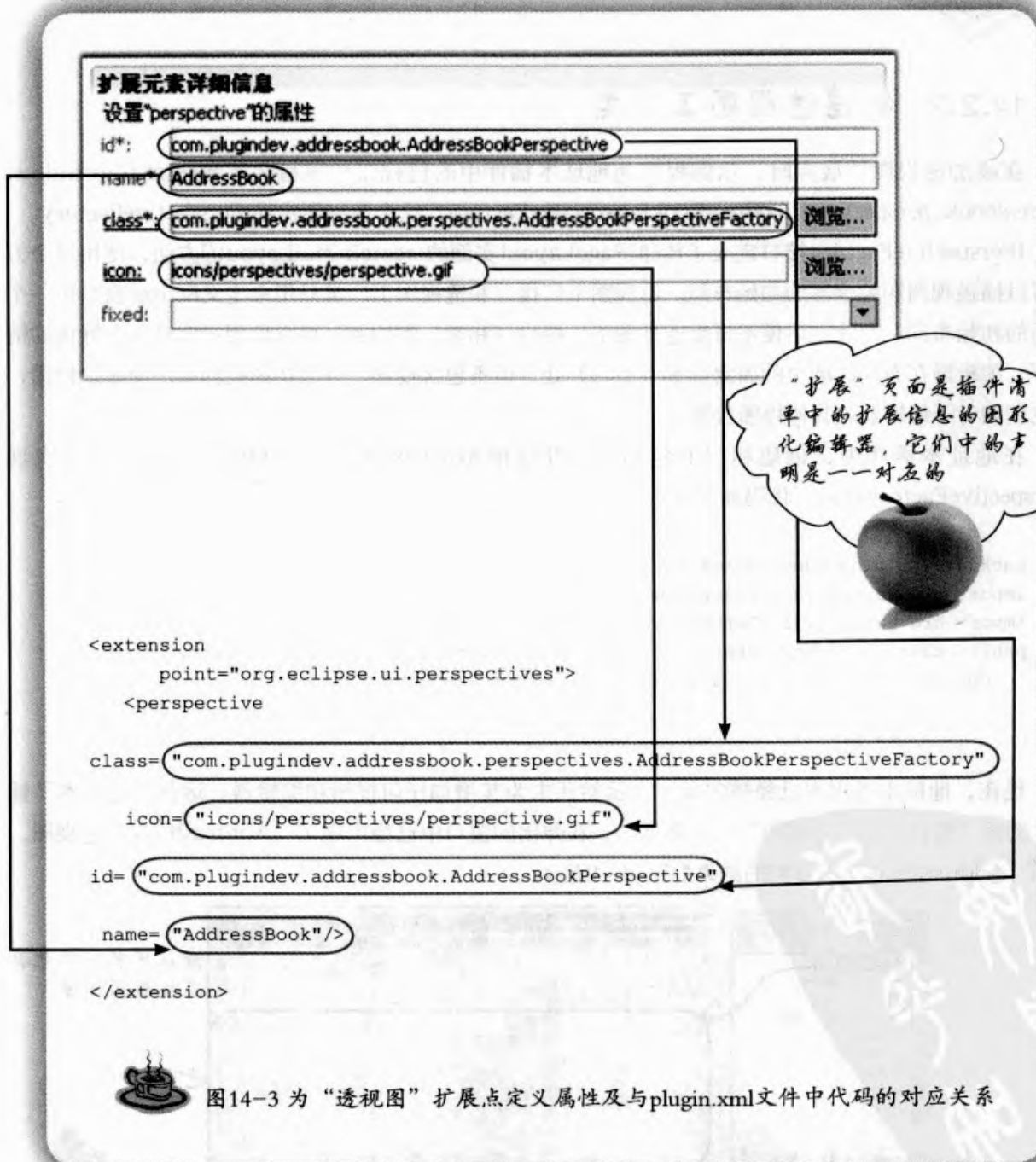


图14-3 为“透视图”扩展点定义属性及与plugin.xml文件中代码的对应关系

★class★——透视图工厂类。该类必须实现org.eclipse.ui.IPerspectiveFactory接口，用来描述同

透视图布局相关的信息。

★icon——和透视图相关联的图标。

注意：



本章的示例程序在icons/文件夹下创建了perspectives目录，并且导入了perspective.gif图标。

14.2.2 创建透视图工厂类

在添加透视图扩展点时，示例程序为地址本插件中的透视图扩展指定了类com.plugindev.addressbook.perspectives.AddressBookPerspectiveFactory，这个类必须实现IPerspectiveFactory。

IPerspectiveFactory接口定义了传递IPageLayout实例的createInitialLayout()方法，使用这个方法可以描述视图和编辑器的初始布局。透视图工厂接口和透视图工厂类只用来定义Eclipse窗口中一个页面的初始布局，在此之后便不再受透视图工厂接口（和类）的约束，可以由用户定制。在默认的情况下，编辑器在布局区域中的固定位置（中心）处，但不包含视图。可以在透视图工厂中添加视图，使它们相对于编辑器或其他视图放置。

在地址本插件中，创建相应的包和类，并且使AddressBookPerspectiveFactory实现IPerspectiveFactory接口，代码如下所示。

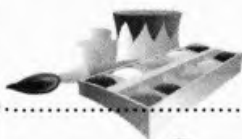
```
package com.plugindev.addressbook.perspectives;
import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;
public class AddressBookPerspectiveFactory implements IPerspectiveFactory {
    public void createInitialLayout(IPageLayout layout) {
    }
}
```

现在，地址本透视图已经被定义了，虽然还未为其增加任何视图和编辑器。运行“地址本”插件，选择“窗口”→“透视图”→“其他”，在弹出的窗口中已经出现了“AddressBook”透视图。选择“AddressBook”，看到的结果如图14-4所示。



图14-4 创建的最简单的透视图

※ 注意 : ※



编辑器区域虽然是默认的，但不是必须的，它可以被隐藏。在某些场合下，可能不希望在透视图中包含编辑器区域，这时应该在透视图工厂中使用 `IPageLayout.setEditorAreaVisible(false)` 来隐藏编辑器区域。



14.3 IPageLayout

`IPageLayout` 是透视图工厂 `createInitialLayout()` 的一个重要参数，该接口定义了支持几乎所有透视图布局所需要的协议，定义了许多非常有用的方法接口。可以用到的方法如下所示。

- ★ `addActionSet(String)`——在页面中添加给定的操作集，参数为操作集的标识符。
 - ★ `addFastView(String)`——在页面布局中添加给定ID的视图，作为一个快速视图。
 - ★ `addFastView(String, float)`——在页面布局中添加给定ID的视图，作为一个快速视图，并且为该快速视图指定宽度比例（相对于工作台）。
 - ★ `addNewWizardShortcut(String)`——在“文件”→“新建”菜单和工具栏“新建”按钮中创建一个新的快捷方式。该快捷方式用来快速指向需要的向导。
 - ★ `addPerspectiveShortcut(String)`——在“透视图”菜单中添加一个透视图快捷方式。
 - ★ `addPlaceholder(String, int, float, String)`——在页面布局中添加给定ID（第一个参数）的视图的占位符。占位符用来在视图未显示之前定义视图的位置。起初视图不可见（除非默认使其可见），当用户打开的视图同给定的视图ID相同时，该视图将出现在同占位符相同的位置。
 - ★ `addShowInPart(String)`——在“显示位置”提示器中增加一项。
 - ★ `addShowViewShortcut(String)`——在“显示视图”菜单中添加一个视图快捷方式。
- 图14-5说明了上述添加方法和为透视图添加的操作的关系。



图14-5 IPageLayout中的添加方法同透视图菜单操作的关系

- ★**addView(String, int, float, String)**——在页面布局中添加给定ID（第一个参数）的视图。
- ★**createFolder(String, int, float, String)**——在页面布局中创建并添加给定ID的文件夹。
- ★**createPlaceholderFolder(String, int, float, String)**——在页面布局中创建并添加给定ID的新文件夹占位符。
- ★**getEditorArea()**——返回该页面布局中的编辑器区域的特定标识符。
- ★**setEditorAreaVisible()**——显示或隐藏页面布局的编辑器区域。
- ★**isEditorAreaVisible()**——指示页面是否显示编辑器区域。
- ★**setFixed(boolean)**——设置当前的布局是否固定。如果当前的布局固定，则既不能移动布局中的部分，也不能放大或缩小，其中的视图也不能被关闭。

★isFixed()——返回当前的布局是否固定。

★getViewLayout(String)——返回指定标识符的视图或占位符的布局。

★addStandaloneView(String, int, float, String)——在页面布局中添加一个单一的视图。所谓单一的视图就是不能和其他视图共用同一位置的视图；而且，除非该视图关闭，否则该视图的标题(Title)不能被隐藏。

※注意：※

除这些方法之外，*IPageLayout*中还包含常用的视图标识。如果需要引用常用视图，可以通过*IPageLayout*来得到它们。



14.4 填充透视图

在为新创建的透视图填充内容之前，首先需要为透视图设计默认页面布局。如图14-6所示，在默认界面中，以编辑器为中心，左侧和下方分别是“地址本”视图和“属性”视图。

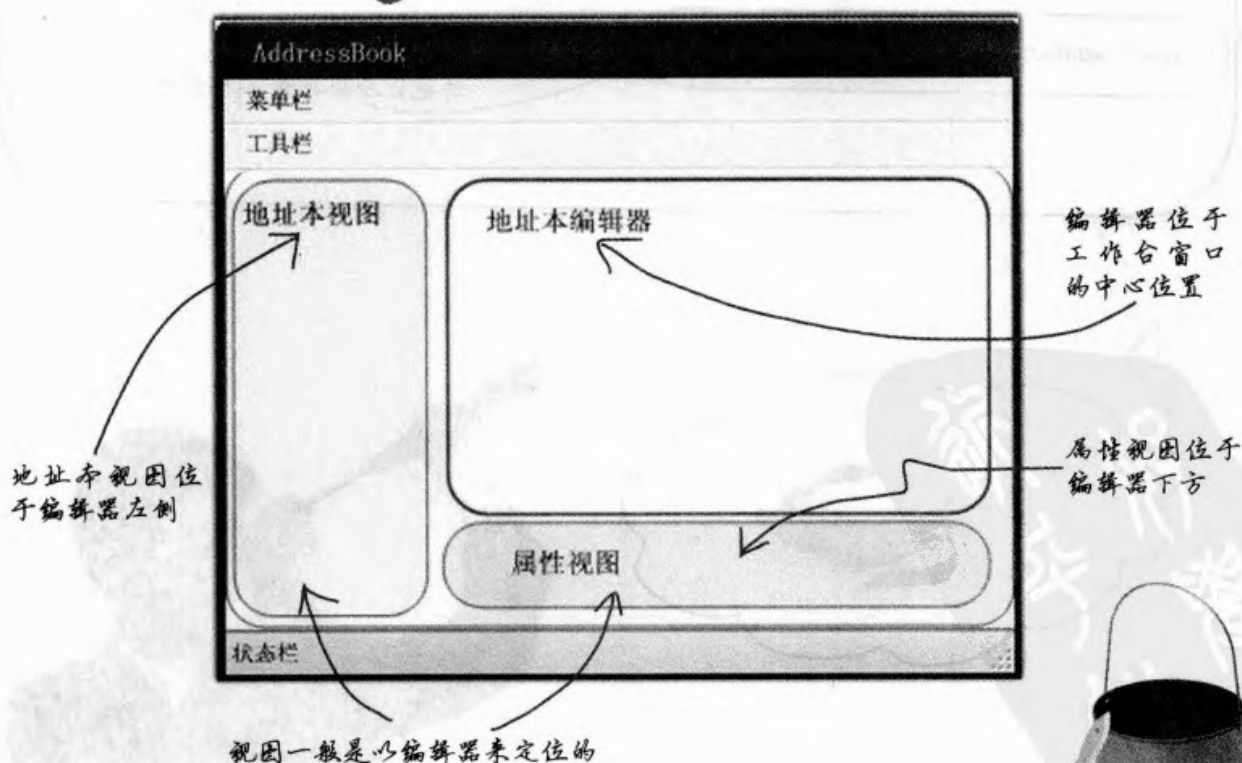


图14-6 设计的透视图默认页面布局



然后，在AddressBookPerspectiveFactory类中加入如下面所示的代码，重新运行地址本插件，打开“AddressBook”透视图，便可以看到图14-6所示的布局。

```
public class AddressBookPerspectiveFactory implements IPerspectiveFactory {
    private static final String ADDRESS_VIEW_ID =
        "com.plugindev.addressbook.views.AddressView";
    private static final String ADDRESS_ACTION_ID =
        "com.plugindev.addressbook.actionSet";
    public void createInitialLayout(IPageLayout layout) {
```

```
String editorArea = layout.getEditorArea();
```

获得编辑器区域

```
layout.addView(ADDRESS_VIEW_ID,
    IPageLayout.LEFT, 0.25f, editorArea);
```

将地址本视图放入编辑器区域的左侧，此处需要提供地址本视图的标识，并且提供一个浮点数字来表示位置

```
IFolderLayout bottom = layout.createFolder("bottom",
    IPageLayout.BOTTOM, 0.66f, editorArea);
```

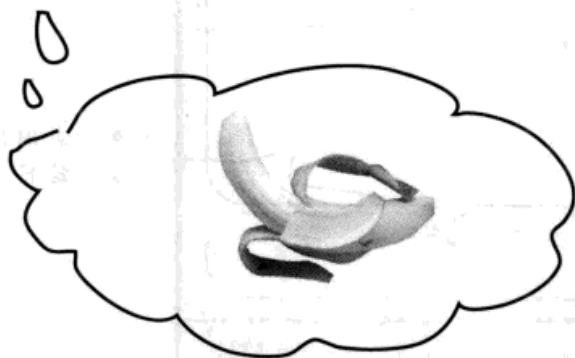
在编辑器区域底部创建一个包布局 (IFolderLayout)

```
bottom.addView(IPageLayout.ID_PROP_SHEET);
bottom.addPlaceholder(IPageLayout.ID_PROP_SHEET);
```

将属性视图放入bottom中

```
layout.addActionSet(ADDRESS_ACTION_ID);
```

将地址本操作集加入透视图



所得到的透视图及其同透视图工厂、插件清单的对应关系如图14-7所示。

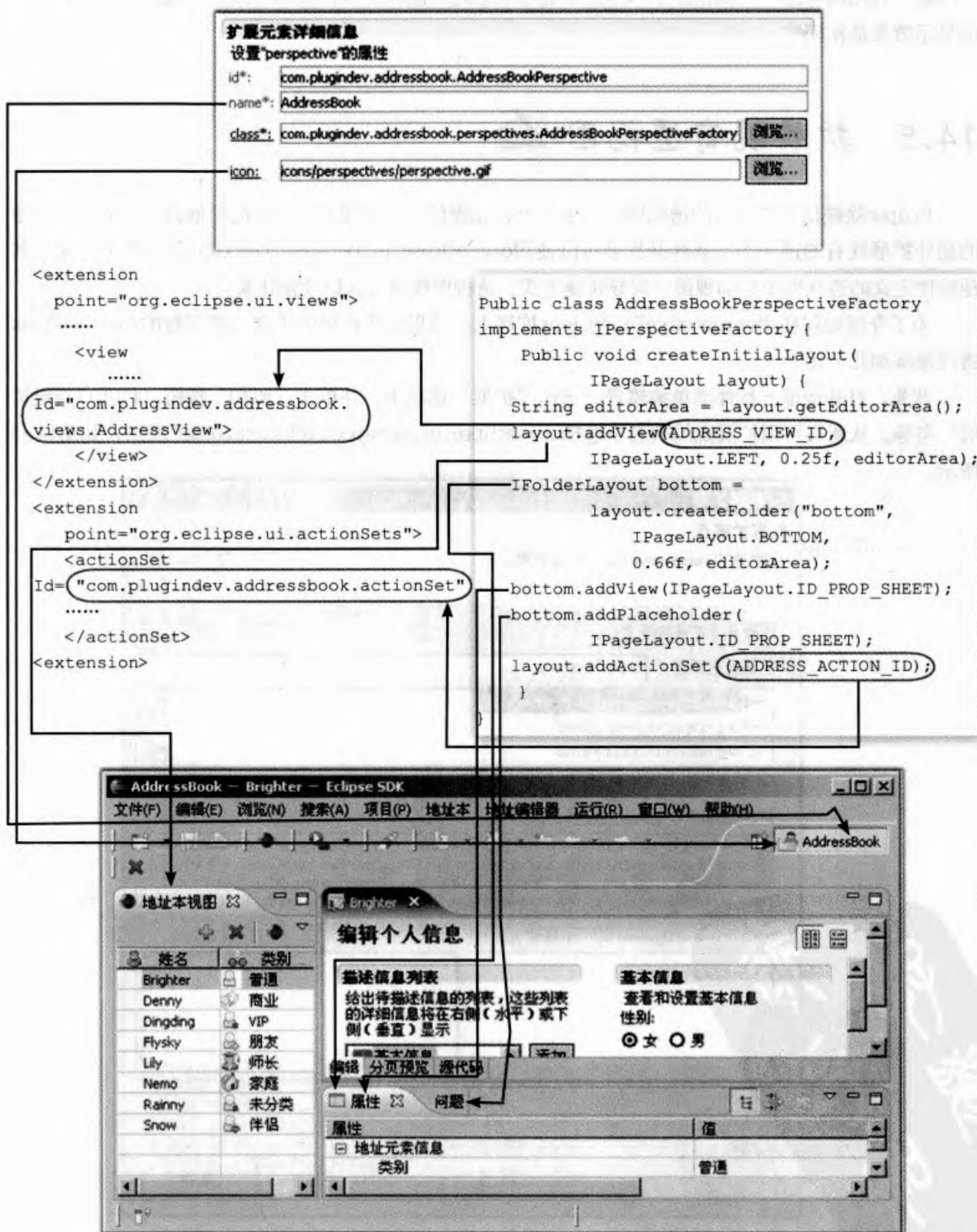


图14-7 透视图及同plugin.xml和透视图工厂插件清单的对应关系

在图14-7中可以看到, 当为“问题”视图添加了视图占位符后, 在属性视图的旁边也出现了“问题”视图的标题。当视图默认不是处于显示状态时(隐藏在其他视图下面), 视图占位符和视图的显示效果是相同的。

14.5 扩展现有透视图

Eclipse除提供了创建新的透视图的扩展点外, 还提供了扩展现有透视图的扩展点, 这方便了其他的插件扩展现有的透视图。插件开发者可以使用`org.eclipse.ui.perspectiveExtensions`扩展点来向其他插件定义的透视图中添加视图、向导快捷方式、透视图快捷方式以及操作集。

为了介绍如何使用`perspectiveExtensions`扩展点, 这里使用扩展的方式为新创建的AddressBook透视图添加几个元素。

首先, 打开地址本插件清单编辑器, 选择“扩展”选项卡, 并单击“添加”按钮, 打开“新建扩展”向导。从所有可用扩展点的列表中选择`org.eclipse.ui.perspectiveExtensions`扩展点, 如图14-8所示。

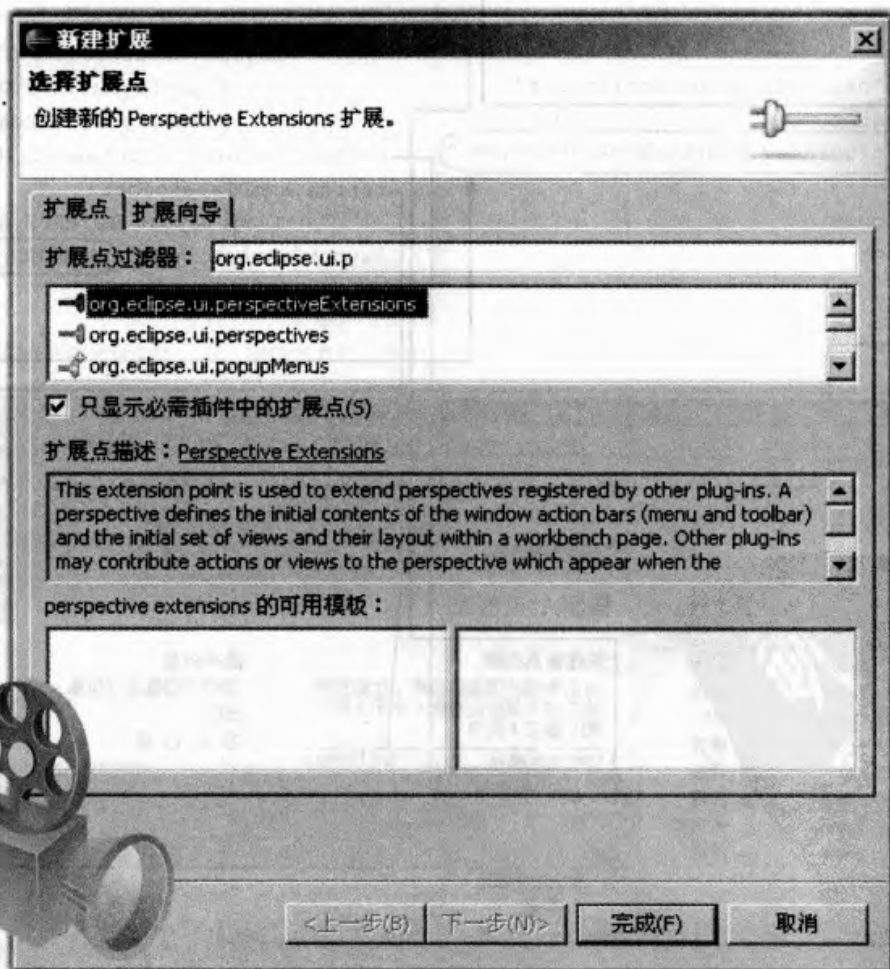


图14-8 选择`org.eclipse.ui.perspectiveExtensions`扩展点

在插件清单编辑器的“扩展”页，右键单击org.eclipse.ui.perspectiveExtensions扩展，并选择“新建”→“perspectiveExtension”菜单。选择后，在右侧的“扩展元素详细信息”部分设置如图14-9所示的属性。该扩展点只有一个属性targetID。

★targetID*——扩展的透视图的标识符。

此列表中为定义perspective可用的所有元素

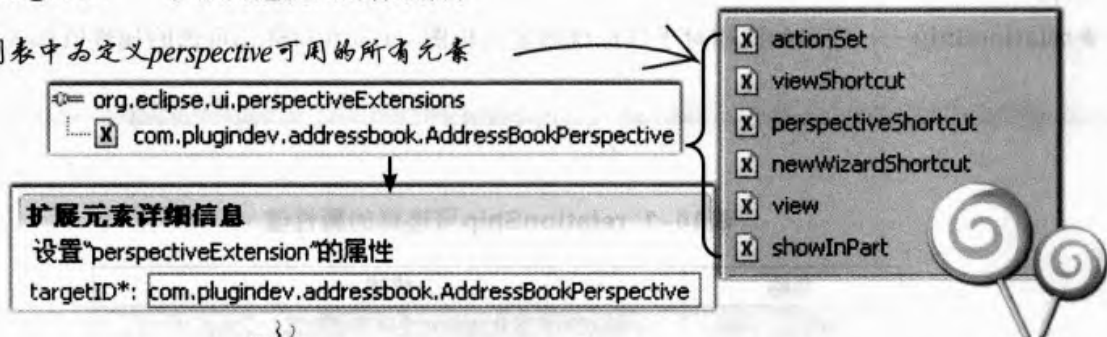


图14-9 设置perspectiveExtension的属性

创建了透视图扩展之后，可以添加很多不同的扩展类型，包括视图、占位符、操作集以及视图和透视图的快捷方式和新向导等。本章将介绍如何为透视图添加视图、占位符，视图和透视图的快捷方式，添加其他元素类似，在此不作介绍。

14.5.1 添加视图和占位符

可以直接在现有透视图中添加视图，也可以添加一个占位符，以便用户打开时，视图出现在正确的位置。本节将为“AddressView”添加“控制台”视图。

在“扩展”页上，右键单击新建的com.plugindev.addressbook.AddressBookPerspective扩展，并选择“新建”→“视图”菜单，然后在右边的“扩展元素详细信息”中输入如图14-10的属性值。

当定义了relative为属性视图，关系为stack之后，控制台视图将会出现在属性视图的右侧

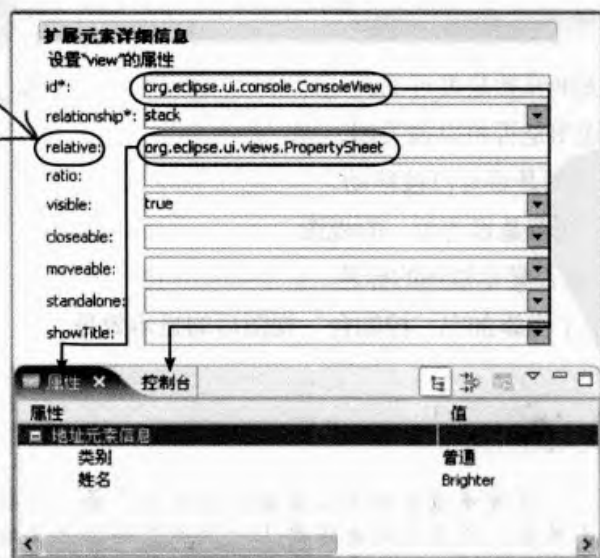


图14-10 显示透视图视图扩展属性的“属性”视图



由图14-10可以看到, view的属性如下所示。

★id——要添加的视图的唯一标识符。本例为“控制台”视图的标识符。

★relative——基准视图的ID。若要相对于某个视图或编辑器来添加其他视图, 需要指定该视图的标识符。

★relationship——应该如何相对于目标视图定向视图。relationship可选的属性值如表14-1所示。

表14-1 relationship可选的属性值

名称	描述
fast	该视图扩展将作为快速视图被创建
stack	该视图扩展将紧贴着relative属性中指定的视图创建。这两个视图在一个Folder布局中
left	该视图扩展将会被放在relative属性中指定的视图左侧 (ratio属性必须被定义)
right	该视图扩展将会被放在relative属性中指定的视图右侧 (ratio属性必须被定义)
top	该视图扩展将会被放在relative属性中指定的视图上侧 (ratio属性必须被定义)
bottom	该视图扩展将会被放在relative属性中指定的视图下侧 (ratio属性必须被定义)

★visible——指示视图刚开始是否可见。

★closeable——指示视图是否可以被关闭。

★moveable——指示视图是否可以被移动。

★standalone——指示视图是否为单一的视图。

★showTitle——指示是否显示视图的标题。

图14-10中同时也显示了当添加完“控制台”视图后的显示效果。

注意



在改变透视图之后重新启动地址本插件, 并不能改变当前透视图的显示, 因为默认会保存上一次用户会话状态。重启时需要清空工作空间数据, 这样在重启后打开AddressBook透视图, 新添加的元素才能够显示。第14.5.2节与此相同。

14.5.2 添加快捷方式

perspectiveExtensions也提供了快速访问相关视图、透视图和新向导的快捷方式。作为示例,本节将在“AddressView”透视图中添加访问“地址本”视图和“AddressBook”透视图的快捷方式。

首先添加一个从“AddressView”透视图访问“地址本”视图的快捷方式。在“扩展”页上,右键单击com.plugindev.addressbook.AddressBookPerspective,在上下文菜单中选择“新建”→“viewShortcut”菜单,然后在右边的“扩展元素详细信息”中输入如图14-11所示的属性值。

viewShortcut元素只有如下一个属性。

★id★——需要添加快捷方式的视图的标识符。

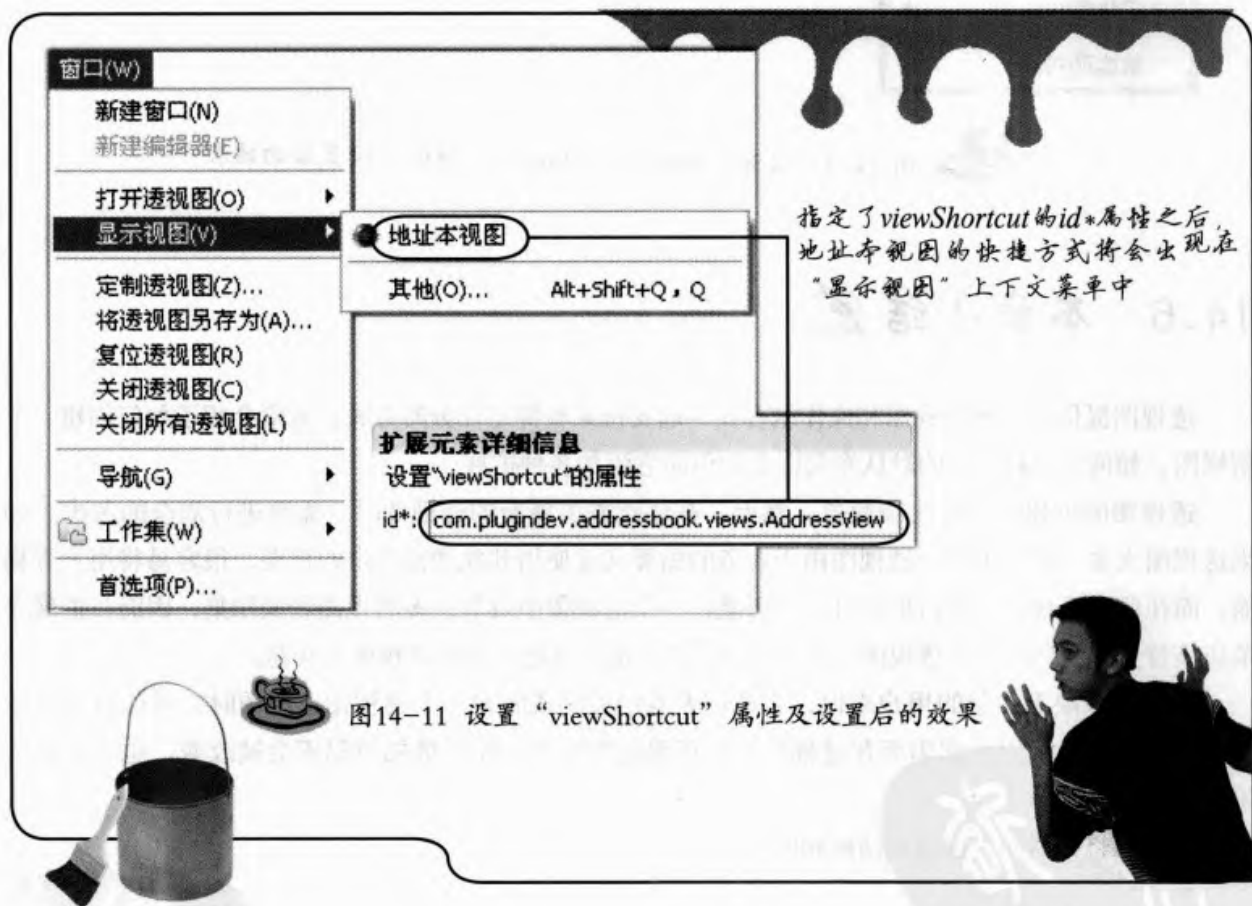


图14-11 设置“viewShortcut”属性及设置后的效果

然后,在“AddressBook”透视图中添加一个访问“AddressBook”透视图的快捷方式。在“扩展”页上,右键单击org.eclipse.ui.resourcePerspective扩展,在上下文菜单中选择“新建”→“perspectiveShortcut”菜单,然后在右边的“扩展元素详细信息”中输入如图14-12所示的属性值。

perspectiveShortcut元素也只有如下一个属性。

★id★——需要添加快捷方式的透视图的标识符。



图 14-12 设置 “perspectiveShortcut” 属性及设置后的效果

14.6 本章小结

透视图提供了一种将视图和操作组合在一起支持某项特定任务的方式。本章介绍了如何创建一个新视图、如何定义新视图的默认布局以及如何向它添加各种扩展。

透视图的创建和扩展比较简单，然而，在创建或扩展新的透视图时，需要进行周全的考虑。如果透视图太多，而且不同的透视图由于业务的需要又要使用其他透视图中的视图，很容易将用户弄糊涂；而在原有透视图上进行扩展时，又可能在一个透视图放置过多的视图和操作集，因而在扩展之前应该首先弄明白向现有透视图添加内容是否合适。这是一个需要权衡的问题。

对于那些缺乏经验的用户来说，当他们无意中关闭或隐藏一个透视图或视图时，可能会变得不知所措。因而，可以根据需要在透视图实现固定的视图，使其最初布局不会被改变，也不会被关闭。

下一章将介绍如何创建对话框和向导。

我们已经实践了,就是如此简单



第15章 对话框和向导 (Dialogs Wizards)

拉开崭新的学习帷幕

进入第15章

上一章介绍了如何定义透视图，将Eclipse插件中的某些视图和编辑器关联起来。本章将针对不同于地址本和编辑器的另外一类Eclipse插件中经常使用的用户交互UI——对话框和向导进行探讨，并将通过完善地址本插件实例，介绍如何在开发插件时合理有效地使用它们。

本章内容包括：

- ★介绍对话框和向导的特点。
- ★介绍SWT和JFace两类不同的对话框。
- ★为插件贡献新的SWT和JFace对话框。
- ★介绍Eclipse向导的体系结构和处理逻辑。
- ★为插件添加新的多页面向导。



15.1 对话框和向导概述

对话框和向导是Eclipse中的第三类用户界面元素，它们同视图和编辑器不同，并不属于Workbench的一部分。相比视图和编辑器，对话框和向导满足了与用户进行受控的交互需求，同时又确保了使用一致的用户界面，它们是Eclipse插件和产品中必不可少的用户界面元素。

对话框和向导通常被看做模态UI，它们限制用户输入请求的信息。模态UI通常用于程序在进行其他处理之前收集和分发信息的场合，并且收集和分发信息的过程不需要同其他的界面元素交互便可以完成；而视图和编辑器是非模态的UI，用户可以自由地同工作台中的所有资源交互，并没有限定用户同它们交互的顺序。图15-1显示了这两种交互模式的不同。

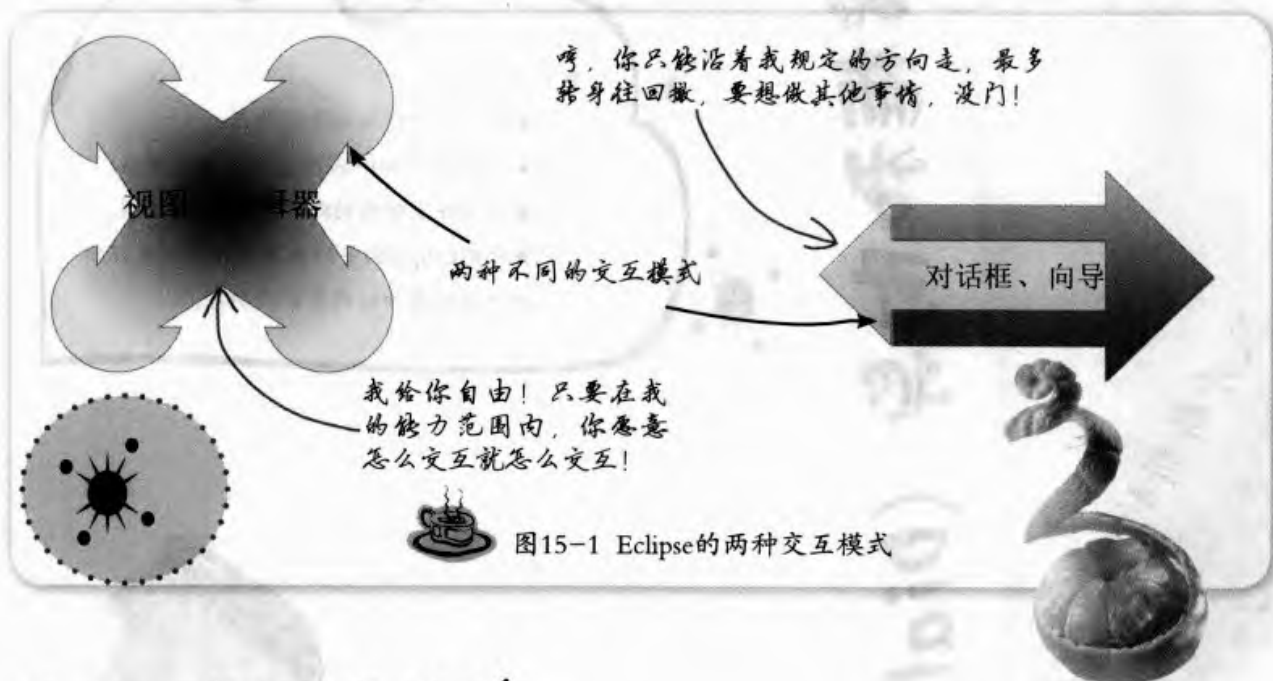


图15-1 Eclipse的两种交互模式

15.2 对话框类别

在介绍如何创建对话框之前，需要区分SWT对话框和JFace对话框，它们是两种不同的对话框层次结构。SWT对话框类（继承了抽象类org.eclipse.swt.widgets.Dialog）为作为SWT插件一部分实现的对话框提供支持，如文件对话框或目录对话框，这些对话框自身就是SWT小部件，是内置平台对话框的Java表示，这意味着在不同的运行平台上，它们的内部实现可能不同。这些对话框为插件提供了与平台相适应的外观，因而，在编写代码时，只需使用这些对话框类即可。

JFace对话框是独立于平台的对话框，它们是向导页面、属性页面、首选项页面的实现基础。JFace支持开发者构建特定的对话框。需要注意的是，Eclipse本身已经提供了一些JFace对话框，在构建自己的专用JFace对话框之前，开发者需要确定现有的JFace对话框能否满足需求，如果不能满足也尽可能地通过扩展现有对话框的功能实现特定的对话框。

本节简单讨论SWT对话框，并提供一个使用SWT对话框的实例，稍后详细介绍JFace对话框。

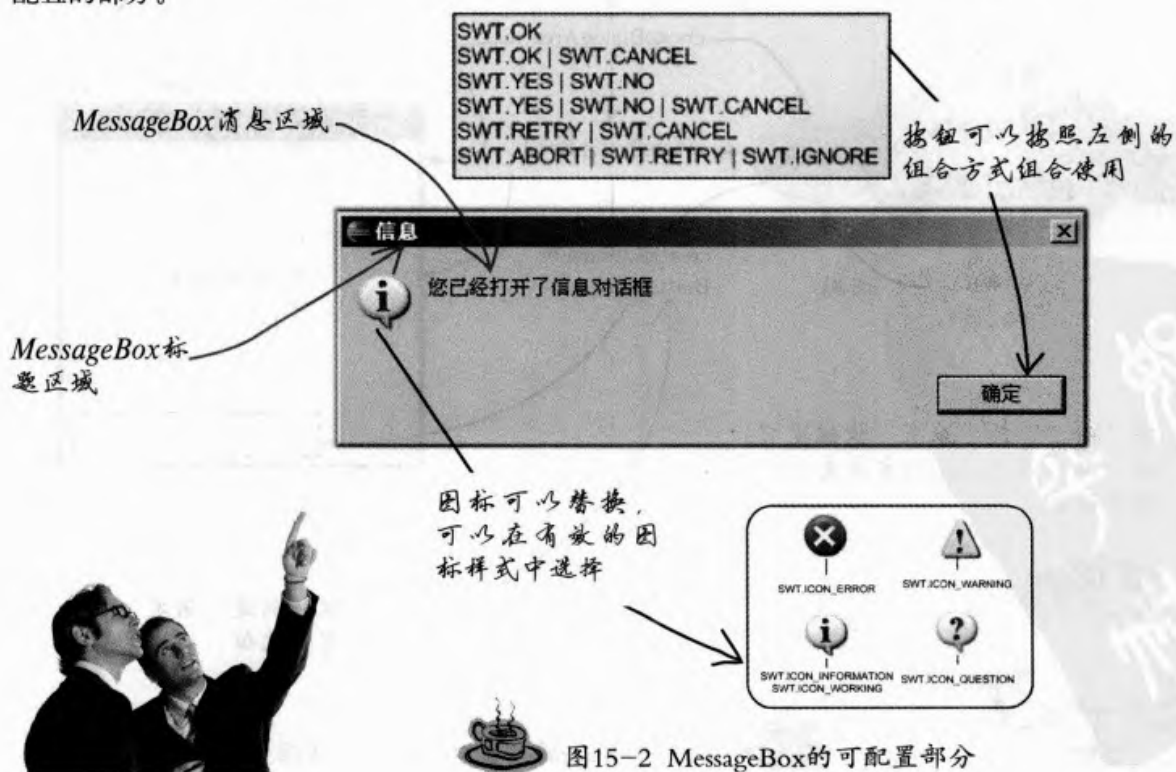
15.2.1 SWT对话框

Eclipse实现的SWT对话框类提供了与底层平台特定对话框的平台无关性接口，它们都位于org.eclipse.ui.swt.widgets包中，对它们的描述如表15-1所示。

表15-1 Eclipse中的SWT对话框

对话框类型	对话框描述
ColorDialog	提示用户从可用颜色的预定义集中选择一种颜色
DirectoryDialog	提示用户浏览文件系统，并选择或新建一个文件夹。有效的SWT样式有： SWT.SAVE 指定新文件夹 SWT.OPEN 选择现有文件夹
FileDialog	提示用户浏览文件系统，并选择或输入一个文件名。有效的样式有： SWT.SAVE 指定新文件 SWT.OPEN 选择现有文件
FontDialog	提示用户从系统所有可用字体中选择一种字体
MessageBox	给用户显示一条消息
PrintDialog	提示用户选择一个打印机和开始打印任务之前的各种相关参数

MessageBox的使用方式非常灵活，可以显示各种不同的消息。图15-2介绍了可以为MessageBox配置的部分。



15.2.2 JFace对话框

org.eclipse.jface.dialogs提供了一些实现标准对话框的类，所有这些对话框类都是JFace抽象类Dialog的子类。Dialog类继承自Window类，二者的生命周期比较相似，不过同Window类相比，Dialog类已经定义好了对话框的基本框架，因而开发者不需要重写createContents()方法来创建其内容，而应该重写createDialogArea(),createButtonBar(),createButtonsForButtonBar()三个方法来构建特定的对话框。如图15-3所示。

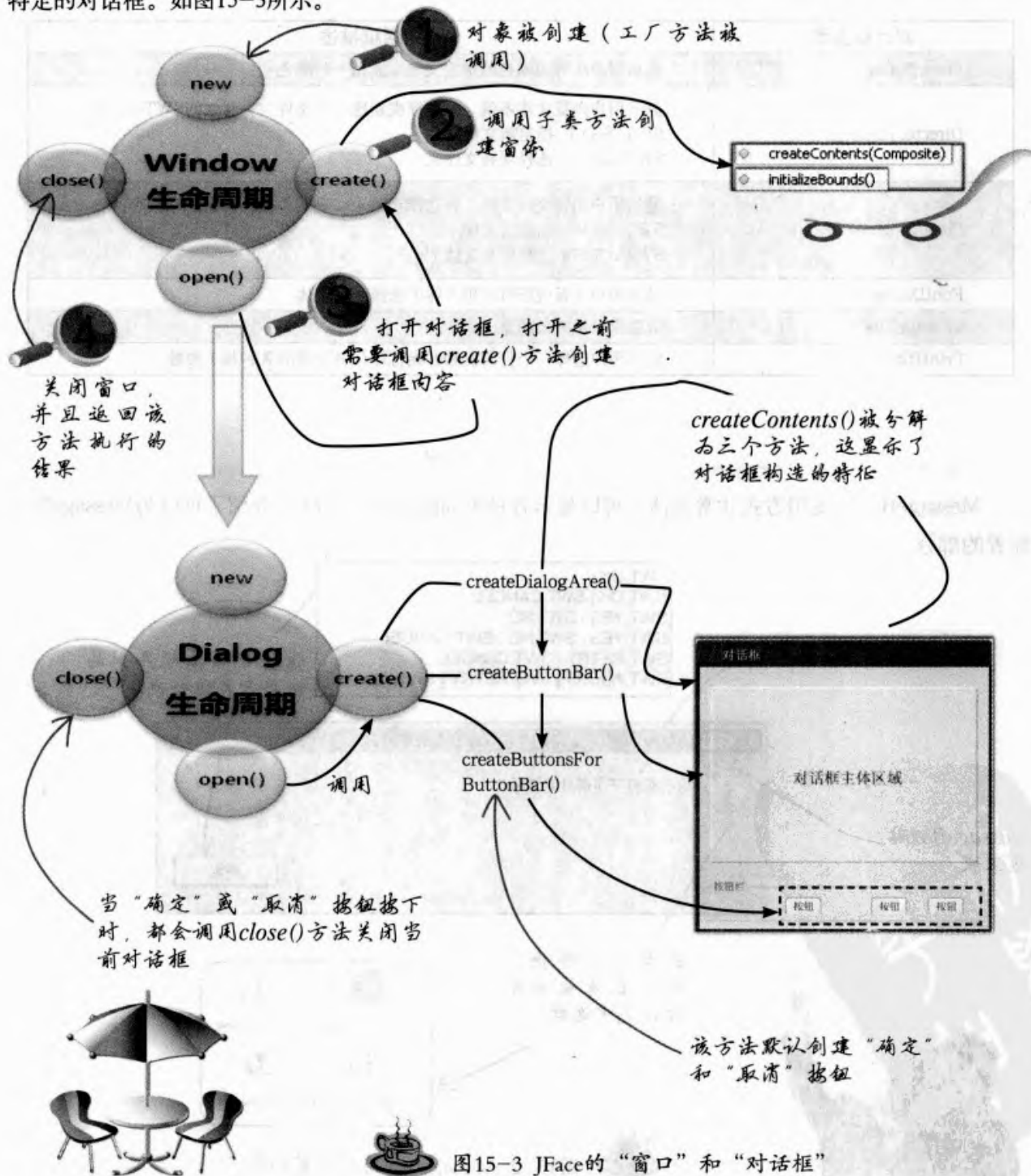


图15-3 JFace的“窗口”和“对话框”

15.2.3 常用JFace对话框

JFace为许多类型的对话框都提供支持，开发者可以重用这些对话框作为自己插件的一部分，它们位于org.eclipse.jface.dialogs和org.eclipse.ui.dialogs包中。

如图15-4所示列出了这两个包提供的几个对话框，该包通过提供这些对话框来支持插件同用户的交互。提供的这些对话框既可以实例化，也可以通过创建子类重用它们

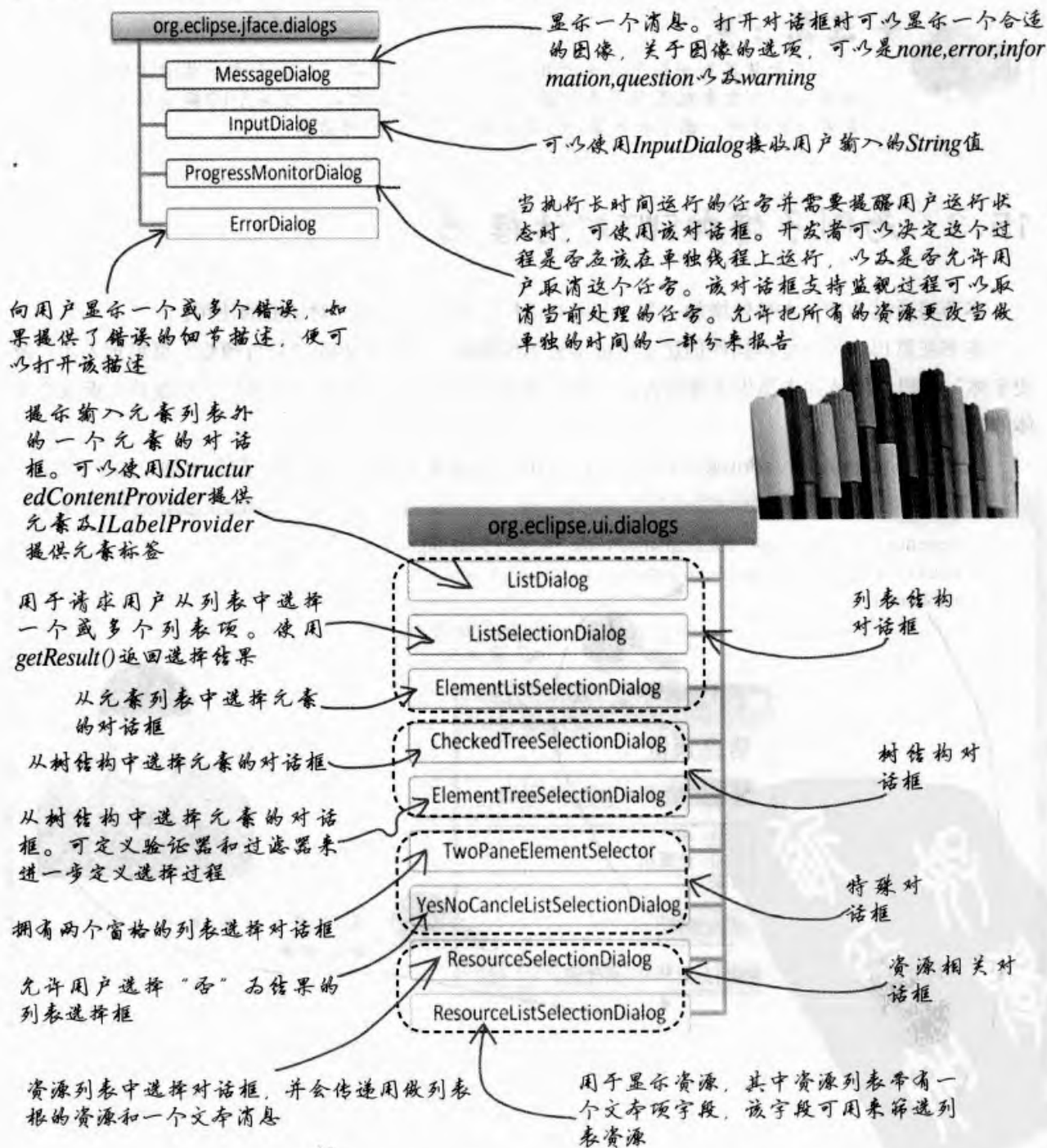


图15-4 org.eclipse.jface.dialogs包提供的对话框

图15-4中列举的对话框都是很容易被重用的对话框，正如在第15.2节中所提到的，在构建自己的专用对话框之前，应该从两个包中寻找是否有可以满足需求的对话框。如果确实需要实现自己的对话框，可以扩展下列类中的一个类。

★org.eclipse.jface.dialogs.Dialog

★org.eclipse.jface.dialogs.TitleAreaDialog

★org.eclipse.ui.dialogs.SelectionDialog



※ 注意 : ※

当希望得到同Eclipse平台相同的外观和感觉时，应该尽可能的重用SWT对话框，例如需要配置颜色与字体、打开文件等操作时，使用SWT现有的对话框是最好的选择。接下来的第15.3节就使用了字体对话框。

15.3 为例子增加SWT对话框

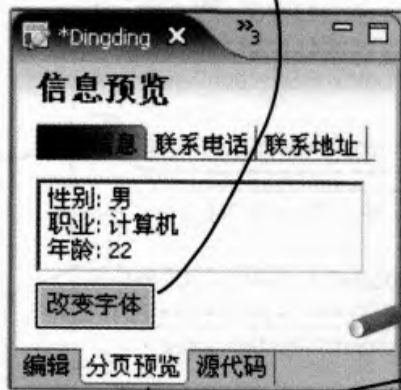
本节将通过为地址本插件增加一个“字体对话框”来介绍对SWT对话框的使用。

本书在第13章为地址本插件创建了地址本表单编辑器，并在其中的“分页预览”页里加入了“改变字体”按钮，但是并未提供实现的方法，现在来为该按钮关联“字体”对话框，实现该页面改变字体的功能。

打开com.plugindev.addressbook.editors包中的PageWithSubPages类，在其中加入如下代码。

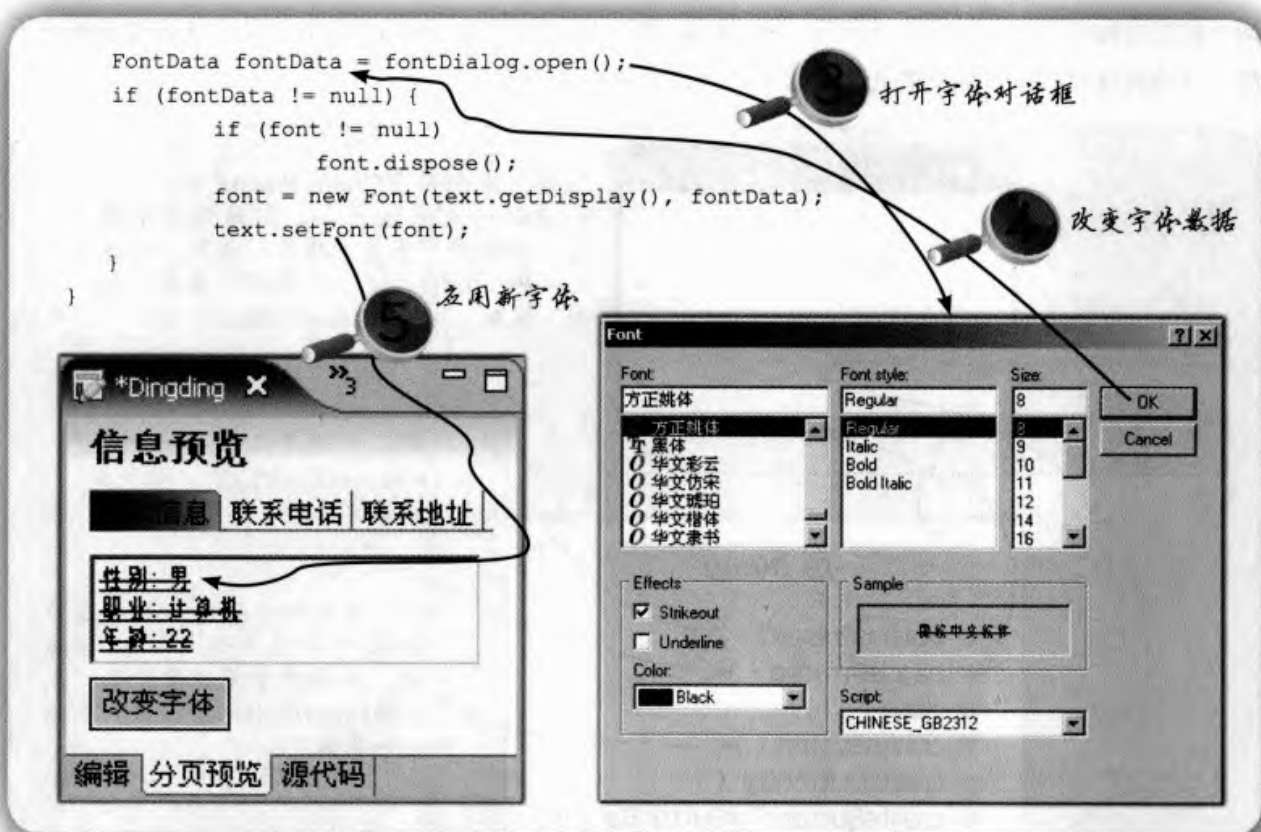
```
fontButton.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
        setFont();
    }
});
```

widgetSelected()
方法被调用



鼠标按下“改变字体”按钮

```
void setFont() {
    FontDialog fontDialog = new FontDialog(getSite().getShell());
    fontDialog.setFontList(text.getFont().getFontData());
}
```



※ 注意 : ※

这里仅为文本设置了字体，虽然“字体”对话框中有颜色选项，但创建的文本并未支持颜色的改变。



由上面代码可以看到，当用户使用鼠标按下“选择字体”按钮后，fontButton中的选择监听器会捕获到部件选择事件，这会触发其中的setFont()方法。示例程序在setFont()方法中创建了SWT对话框FontDialog的实例，并通过该实例获得系统安装的字体信息。当用户选择新的字体，单击“OK”按钮后，新的字体信息将会返回。通过为文本对象设置新字体，文本框中的字体会得到更新。

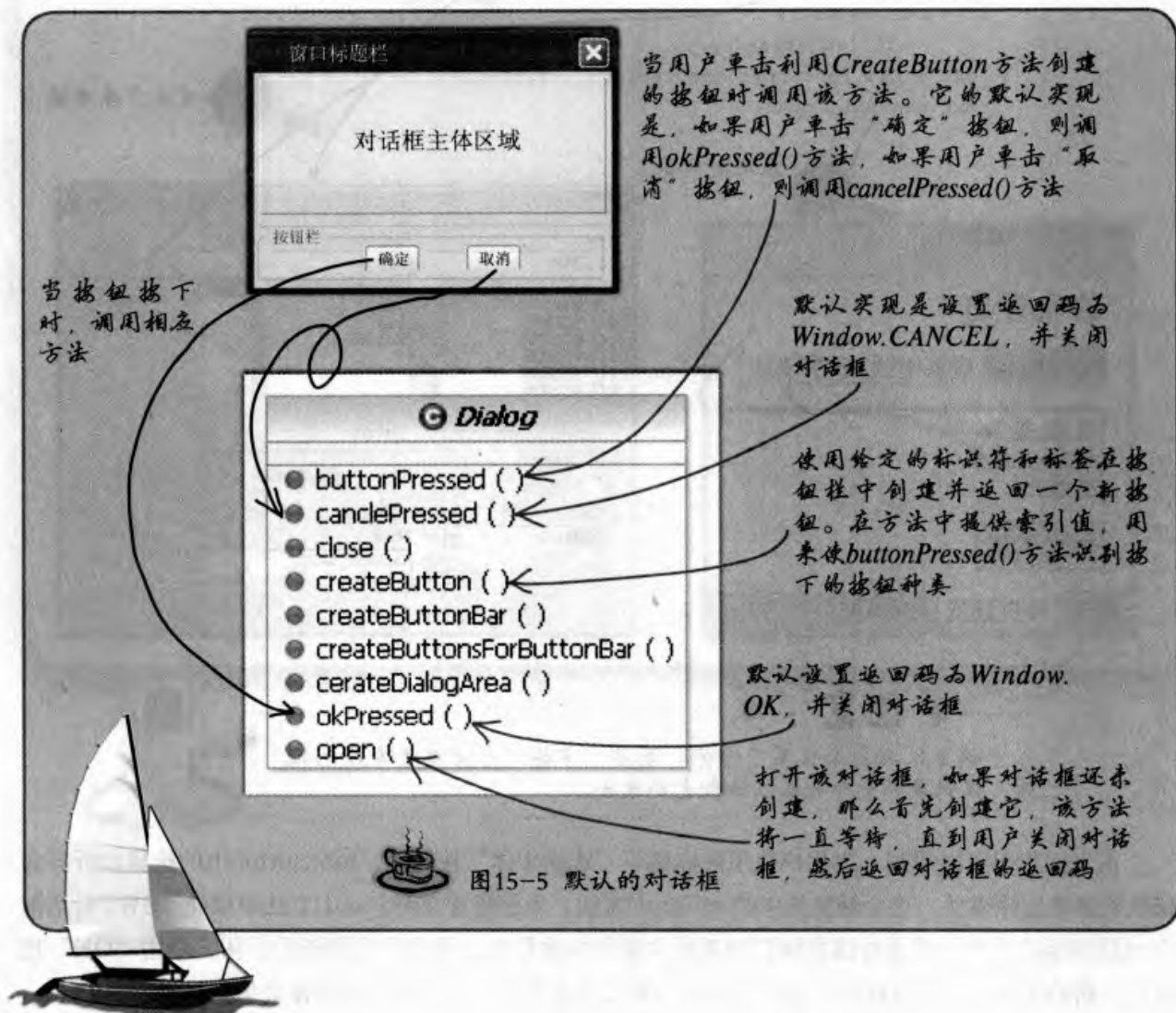
15.4 创建JFace对话框

本节将扩展org.eclipse.jface.dialogs.Dialog类改善本书在第12章创建的类别过滤器。在第12章创建的过滤器使用了JFace对话框中的InputDialog，仅支持用户输入字符串来匹配类别，而一般用户可能并不了解地址本插件为他们提供的分类。本节将为“地址本”视图创建一个专门的对话框，通过该对话框向用户展示基于姓名、类型的过滤内容选项。

15.4.1 使用Dialog类创建JFace对话框

org.eclipse.jface.dialogs.Dialog类默认创建了一个包含了内容区域和拥有“确定”、“取消”

两个按钮的按钮栏，如图15-5所示。扩展抽象类Dialog创建JFace对话框的方法是，先创建该类的子类，并为特殊目的覆盖该类的几个方法来定制该对话框。



15.4.2 为“地址本”视图创建过滤器对话框

回顾一下第12.4.7节，示例程序为视图创建了一个过滤器对话框，它使用了JFace提供的输入对话框，允许用户在其中输入一个表示类别的字符串并提交结果，这样只能按照某个类别过滤，不属于该类别的地址项只能消失。由于地址本插件默认提供的类别是固定的，因而可以为用户提供更为直观的方式在对话框中选择。此外，最好过滤器支持对名称的过滤，使得用户可以通过两种方式快速找到所需的地址项，这就需要重新设计对话框，满足这些要求。

在地址本插件项目中，新建com.plugindev.addressbook.dialogs包，在其中创建AddressViewFilterDialog（扩展org.eclipse.jface.dialogs.Dialog）。

首先设置namePattern和selectedCategories，表明要过滤的名称样式和需要在地址本插件中显示的类别，建立工厂方法初始化变量，代码如下所示。



建立工厂方法，
初始化变量



```
private String namePattern;
private Collection<AddressCategory> selectedCategories;
public AddressViewFilterDialog(Shell parentShell,String
namePattern,
AddressCategory[] selectedCategories) {
super(parentShell);
this.namePattern = namePattern;
this.selectedCategories = new HashSet<AddressCatego
ry>();
for(int i = 0; i < selectedCategories.length; i++){
this.selectedCategories.add(selectedCategori
es[i]);
}
}
```

然后，需要覆盖createDialogArea()方法来创建对话框的显示内容，如下所示。



创建对话框显示

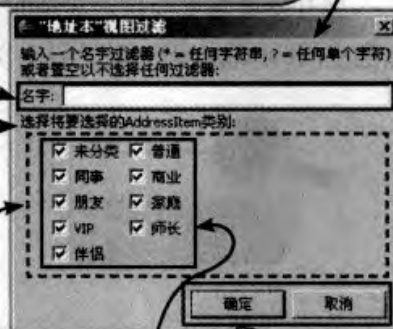
```
final Label filterLabel = new Label(container, SWT.
NONE);
filterLabel.setText("输入一个名字过滤器(* = 任何字符串 "
+ "? = 任何单个字符" + "\n或者置空以不选择任何过滤器");
```

```
namePatternField = new Text(container, SWT.BORDER);
final Label typesLabel = new Label(container, SWT.NONE);
typesLabel.setLayoutData(new
typesLabel.setText("选择将要选择的AddressItem类别:");
```

```
final Label typesLabel = new Label(container, SWT.
NONE);
typesLabel.setText("选择将要选择的AddressItem类别:");
```

```
final Composite checkboxComposite = new Composite(container, SWT.NONE);
```

```
private void createCheckboxes(Composite parent){
categoryFields = new HashMap();
AddressCategory[] allCategories = AddressCategory.getTypes();
for (int i = 0; i < allCategories.length; i++) {
final AddressCategory category = allCategories[i];
final Button button = new Button(parent, SWT.CHECK);
button.setText(category.getCategoryName());
categoryFields.put(category, button);
button.addSelectionListener(new SelectionAdapter() {
public void widgetSelected(SelectionEvent e) {
if (button.getSelection())
selectedCategories.add(category);
else
selectedCategories.remove(category);
}
});
}
}
```



JFace对话框默
认提供的按钮

根据类别的
多少来创建
CheckBox，
在地址本插
件中定义
了9个类别，
因而创建
出了9个
CheckBox

监听用户的选择，当在CheckBox
前打勾时就将该CheckBox代表的
类别加入selectedCategories

在创建完对话框的显示之后, 需要为显示的窗体控件提供初始化内容, 初始化内容表示了在本插件应用程序刚开始启动时的过滤设置, 如下所示。

为控件创建
初始化内容

为“内容过滤”
文本框创建初始
化内容

为“类别过滤”
选择项创建初始
化内容

```
private void initContent(){
    namePatternField.setText(namePattern != null ? namePattern :
        "");
    namePatternField.addModifyListener(new ModifyListener() {
        public void modifyText(ModifyEvent e) {
            namePattern = namePatternField.getText();
        }
    });

    AddressCategory[] allCategories = AddressCategory.getTypes();
    for (int i = 0; i < allCategories.length; i++) {
        AddressCategory eachType = allCategories[i];
        Button button = (Button) categoryFields.get(eachType);
        button.setSelection(selectedCategories.contains(eachType));
    }
}
```

当开发者需要为自己定制的对话框定义名称时, 需要使用org.eclipse.jface.window.Window抽象类中的configureShell()方法, 为其指定新Shell, 代码如下所示。

设置对话框标题

```
protected void configureShell(Shell newShell){
    super.configureShell(newShell);
    newShell.setText("\ "地址本\ "视图过滤");
}
```

现在, 过滤器对话框已经创建好了, 但是, 原来定义的过滤规则已经不再适用于新的过滤器了。细心的读者可能已经发现, 至少有三个方面的问题需要解决, 如图15-6所示。



•地址本插件现在还没有针对名字的过滤器, 如何使地址本插件支持名称过滤呢?

名称过滤

•原来使用字符串匹配来决定类别过滤, 现在使用了CheckBox, 怎么办?

类别过滤

•虽然对话框已经创建好了, 可是如何在运行时打开它?

打开对话框



图15-6 定制完对话框后需要解决的三个问题

•地址本插件现在还没有针对名字的过滤器，
如何使地址本插件支持名称过滤呢？

名称过滤



下面就针对这三个问题——提供解决方案。

回答是：为地址本插件增加针对名称的过滤器，使得当在“名字”文本框中输入字符串后，地址本插件能根据输入的内容决定过滤的地址项。

只要在新的对话框中创建一个文本框，在该文本框中输入需要过滤的名称匹配项，因而名称过滤器的实现同第12章介绍的类别过滤器的实现机制基本相同，可以参看第12.4.7节“视图的过滤”中介绍的类别过滤器的创建过程，在此仅回顾一下处理的过程，如图15-7所示，不再列出具体的代码（查看具体源代码请参见随书获赠光盘第14章源代码）。

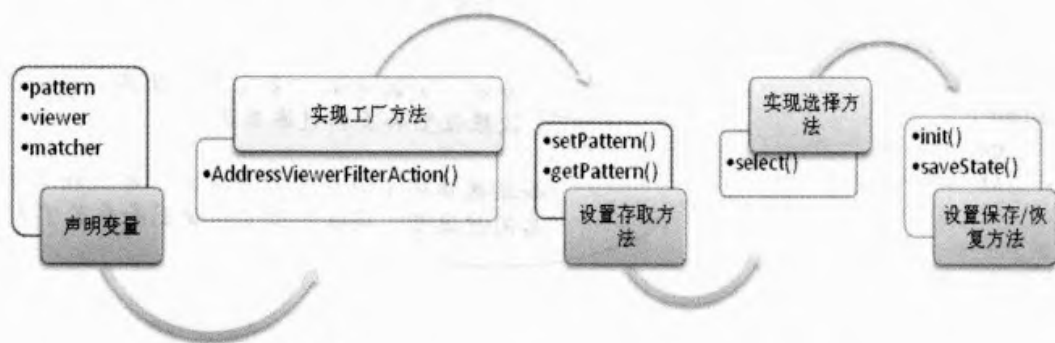


图15-7 “名称”类别过滤器的创建过程

•原来使用字符串匹配来决定类别过滤，现在使用了CheckBox，
怎么办？

类别过滤



回答是：需要修改类别过滤器类来支持CheckBox。

首先，将只能提供字符串存取器方法的getPattern()和setPattern()方法修改为可以获得AddressCategory（参见第12章“视图”中对AddressCategory的介绍）的存取器方法，实现如下所示。

创建一个保存地址类别的字段

```
private HashSet categories;

public AddressCategory[] getCategories() {
    if (categories == null)
        return AddressCategory.getTypes();
    return (AddressCategory[]) categories.toArray(
        new AddressCategory[ categories.size() ] );
}
```

提供获得地址类别的方法，用于向过滤器对话框传递需要过滤的地址类别项



为类别过滤器提供设置地址本
类别的方法

```
public void setCategories(AddressCategory[] selectedTypes) {
    AddressCategory[] allCategories = AddressCategory.getTypes();
    boolean filtering = categories != null;
    if (selectedTypes.length < allCategories.length) {
        categories = new HashSet();
        categories.addAll(Arrays.asList(selectedTypes));
        if (!filtering)
            viewer.addFilter(this);
        else
            viewer.refresh();
    }
    else {
        categories = null;
        if (filtering)
            viewer.removeFilter(this);
    }
}
```

还没设置类别过滤器，就为地址本
查看器增加该过滤器

否则，刷新地址本查看器，使其
按照设置的类别过滤器显示

如果选择的类别等于所有类别，那么相当于没有
启用过滤器，将该过滤器从视图查看器中去除



然后，修改select()方法，使其能够对AddressItem正确过滤，代码如下所示。

```
public boolean select(Viewer viewer, Object parentElement, Object element) {
    return categories == null
        || categories.contains(((AddressItem) element).getCategory());
}
```

最后，更改与保存和读取相关的saveState()和initState()方法，保证用户在关闭和再次打开“地址本”视图时得到前后一致的体验。代码如下所示。

```
public void saveState(IMemento memento) {
    if (categories == null)
        return;
    IMemento mem = memento.createChild(TAG_CATE);
    int index = 0;
    for (Iterator iter = categories.iterator(); iter.hasNext();) {
        mem.putString(
            TAG_CATE_ID + index,
            ((AddressCategory) iter.next()).getCategoryName());
        index++;
    }
}
```

在每个TAG_CATE_ID后面加入
index用来区分不同的类别

使用系统提供的XML
快照保存当前状态





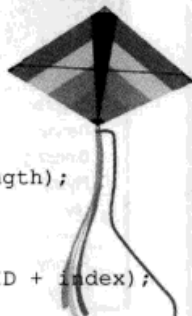
从系统的XML快照中获取最近一次的过滤信息

根据XML快照中获取的信息获得相应的类别对象

将选中的类别对象放入someTypes中

根据someTypes中拥有的类别调用setCategories()方法设置过滤类别

```
public void init(IMemento memento) {
    AddressCategory[] allTypes = AddressCategory.getTypes();
    IMemento mem = memento.getChild(TAG_CATE);
    if (mem == null) {
        setCategories(allTypes);
        return;
    }
    List someTypes = new ArrayList(allTypes.length);
    int index = 0;
    while (true) {
        String eachId = mem.getString(TAG_CATE_ID + index);
        if (eachId == null)
            break;
        AddressCategory eachCategory =
            AddressCategory.getCategoryByName(eachId);
        if (eachCategory != null)
            someTypes.add(eachCategory);
        index++;
    }
    setCategories((AddressCategory[]) someTypes
        .toArray(new AddressCategory[ someTypes.size() ]));
}
```



•虽然对话框已经创建好了,可是如何在运行时打开它?

打开对话框



回答是: 在运行时打开定义的对话框, 只要修改类AddressViewerFilterAction (位于com.plugindev.addressbook.actions插件包中) 在其中添加同nameFilter相关的代码, 并且修改run()方法即可。代码如下所示。

```
public void run(){
    AddressViewFilterDialog dialog = new AddressViewFilterDialog(shell,
        nameFilter.getPattern(),
        categoryFilter.getCategories());

    if(dialog.open() != AddressViewFilterDialog.OK)
        return;
    nameFilter.setPattern(dialog.getNamePattern());
    categoryFilter.setCategories(dialog.getSelectedCategories());
}
```

声明AddressViewFilterDialog对话框, "new" 将会调用工厂方法初始化相关字段

dialog.open() 将会调用 createContents() 方法创建对话框内容, 并且一直作用到对话框关闭, 关闭后返回对话框关闭时的状态

※ 注意: ※

由于同时为地址本插件设置了两个过滤器, 因而只有同时符合这两个过滤器要求的内容才会被显示。



至此，地址本过滤对话框便可以正常使用了。图15-8显示了地址本过滤对话框的运行效果。

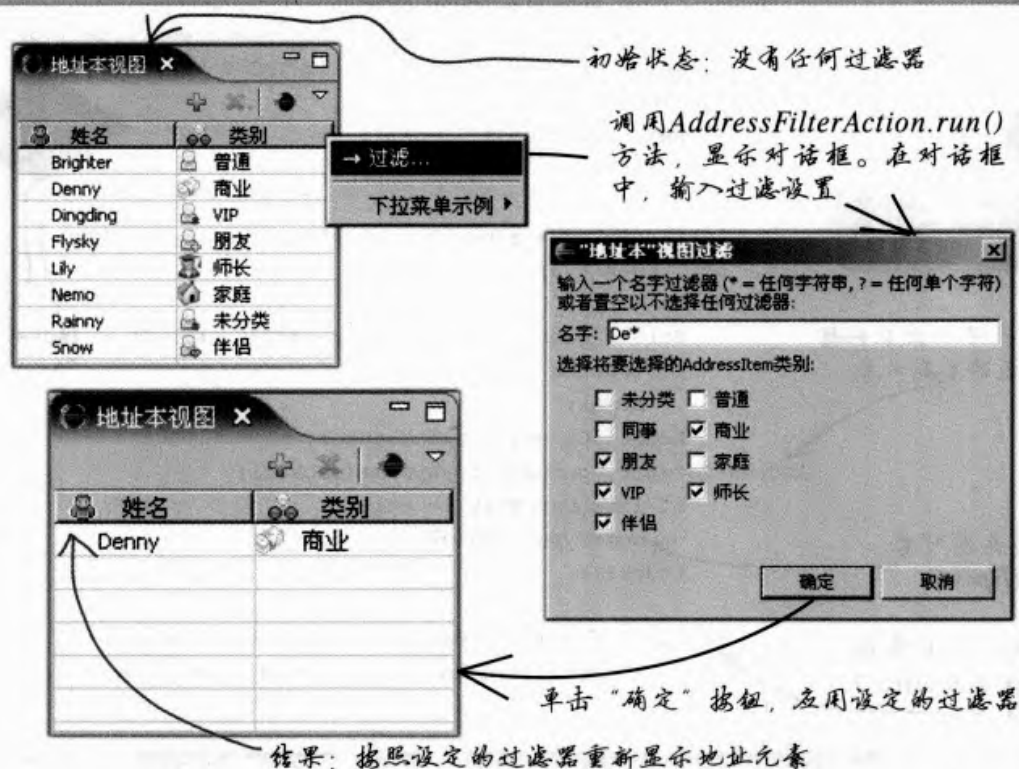


图15-8 地址本过滤器对话框的运行效果

15.5 向导介绍

一个对话框中所能容纳的信息是有限的。当要提供许多信息或需要一个特定的顺序来收集信息时，就需要使用向导 (Wizard) 了。向导由几个不同的底层部件构成，如图15-9所示。

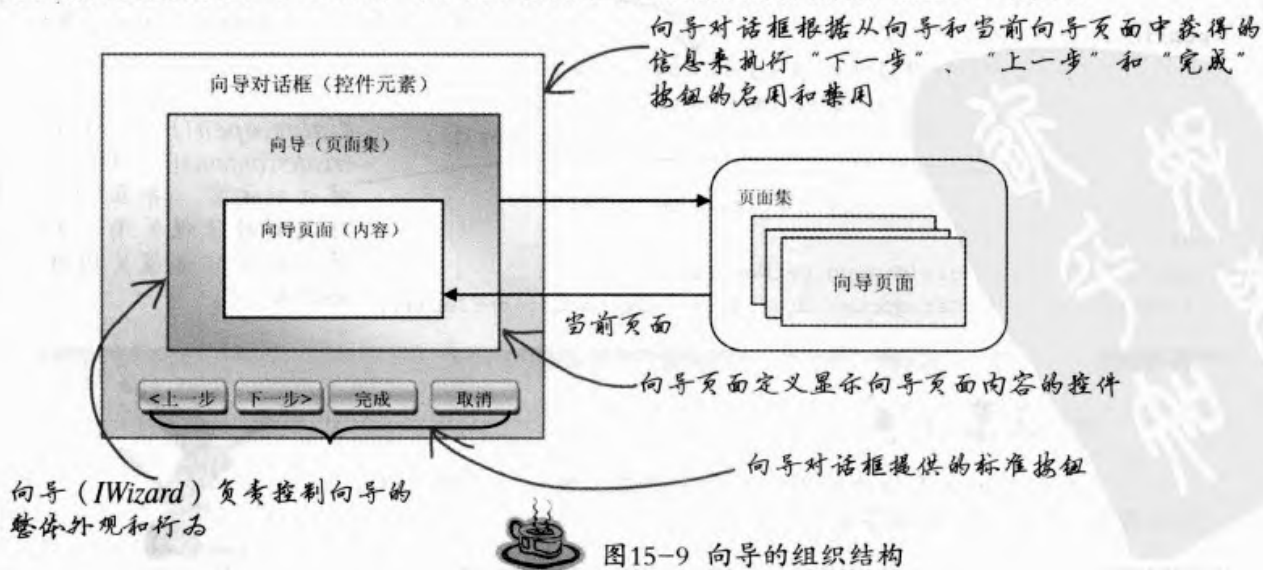


图15-9 向导的组织结构

15.5.1 向导对话框

向导对话框 (org.eclipse.jface.wizard.WizardDialog) 是Dialog的一个专门子类, 它作为向导中的顶级对话框, 定义标准按钮并管理为它提供的一组向导页面。向导对话框的顶部是标题区, 中间是一个显示向导页面的内容区域, 如图15-10所示。

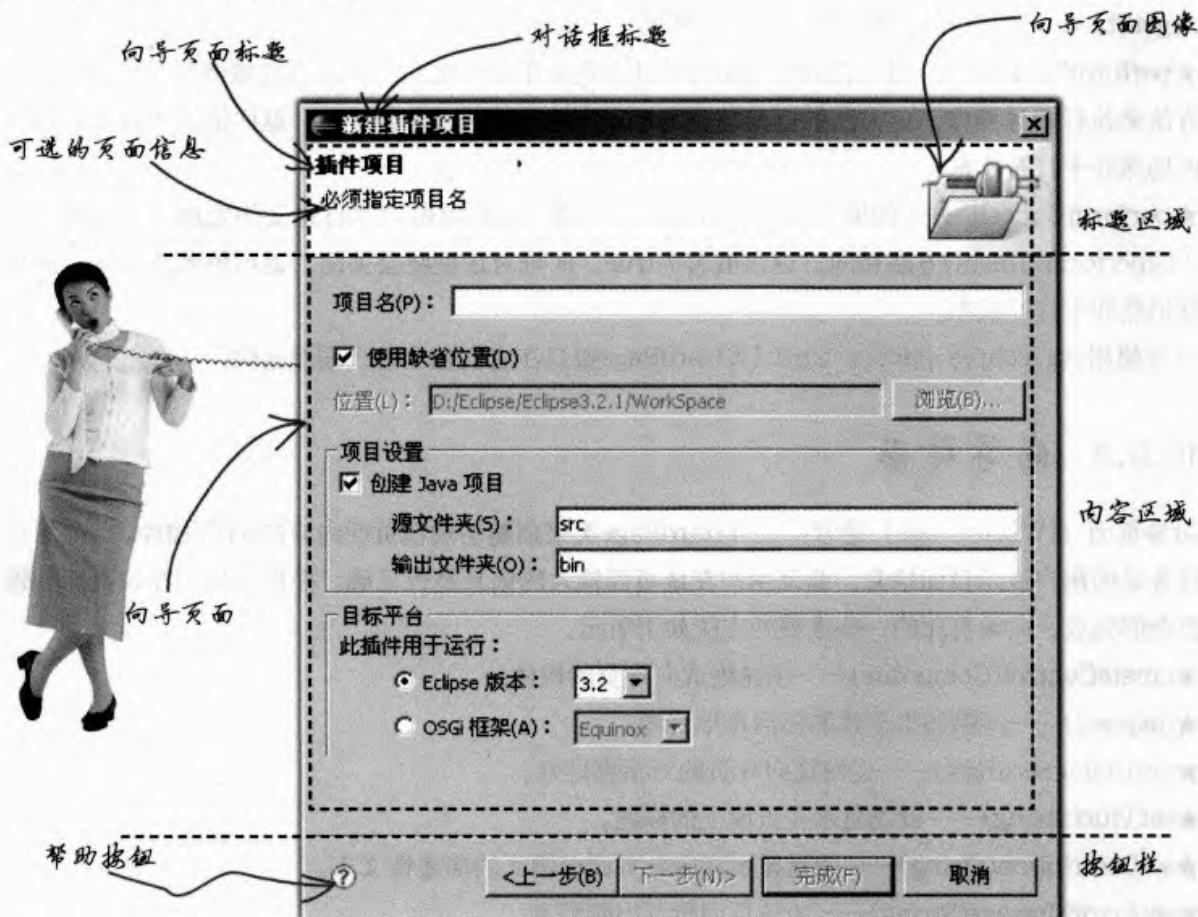


图15-10 向导对话框的结构 (以“新建插件项目”向导为例)

向导对话框使用IWizard接口来获得将要显示的页面, 并向向导通知用户的交互。

15.5.2 向导

向导 (IWizard) 负责控制向导的整体外观和行为, 例如, 标题栏文本、图像以及帮助按钮的可用性。向导通常使用相应的 DialogSettings 来获取 (和存储) 向导页面上的控件设置的默认值。Wizard类提供了向导的默认实现, 通常通过扩展Wizard类来实现特定向导的行为。向导的主要任务包括创建向导页面并将它们添加到向导中; 实现当用户按“完成”按钮时应该发生的行为。

Wizard类中的比较重要的方法如下所示。

- ★addPages()——子类应该覆盖该方法, 通过调用addPage()添加适当的页面。
- ★dispose()——清除该类创建的任何本地资源, 如图像等。

★**getDialogSettings()**——返回该向导的对话框设置。

★**getPreviousPage(IWizardPage)**——返回在特定向导页前面显示的向导页，如果没有则返回null。

★**getNextPage(IWizardPage)**——返回在特定向导页面后面显示的向导页，如果没有则返回null。

★**getStartingPage()**——获得向导中显示的第一个页面。

★**performFinish()**——当“完成”按钮可用并且用户单击该按钮时，该方法被调用。子类应该覆盖该方法来执行向导操作。该方法的返回值若为true，向导对话框将会关闭，返回值若为false，向导对话框仍然处于打开状态。

★**performCancel()**——如果子类需要当单击“取消”按钮时做一些自定义的处理，需要覆盖该方法。同performFinish()方法相同，返回值若为true，向导对话框将会关闭，返回值若为false，向导对话框仍然处于打开状态。

向导使用org.eclipse.jface.wizard.IWizardPage接口在它们包含的页面间通信。

15.5.3 向导页面

向导页面 (IWizardPage) 通常扩展 WizardPage 类来创建构成该页面内容的 SWT 控件。向导页面的任务是向用户展示页面信息、验证用户在该页面输入的信息是否正确，并指示用户在该页面的输入是否全部完成。向导页面的一些重要的方法如下所示。

★**createControl(Composite)**——创建组成向导页的控件。

★**dispose()**——清除该类创建的任何本地资源。

★**getDialogSettings()**——返回该向导页的对话框设置。

★**setTitle(String)**——设置显示在页面上的标题。

★**setDescription(String)**——设置显示在向导页面标题区的描述性文本。

★**setErrorMessage(String)**——为该页面设置出错消息。

★**setImageDescriptor(ImageDescriptor)**——设置向导页面的图像。

★**setMessage(String)**——设置该向导页面的信息。

★**setPageComplete(String)**——设置该页面是否已完成。如果该页面已完成，则“下一步”按钮（如果该页面不是最后一页）和“完成”按钮将会启用。

★**setVisible(boolean)**——设置该对话框页的可见性。子类可以扩展该方法来检测页面何时变为活动页面。

15.6 添加向导

本节将为地址本插件添加一个新的向导。添加向导仍然需要在插件清单中定义向导扩展项，可以使用Eclipse提供的三个扩展点来实现向导扩展项。这三个扩展点以及同它们相对应的接口名称如表15-2所示。

本节将使用org.eclipse.ui.newWizard扩展点来为地址本插件创建一个“新建地址元素”的向导，并且完善第12章介绍的“添加”按钮的操作。

表15-2 用于向用户界面添加向导的扩展点

扩展点	接口
org.eclipse.ui.newWizard	INewWizard
org.eclipse.ui.importWizard	IImportWizard
Org.eclipse.ui.exportWizard	IExportWizard

创建一个可运行的向导需要经过下列步骤。

- 1 定义一个向导扩展；
- 2 实现一个向导；
- 3 实现向导页面；
- 4 添加向导处理逻辑。

下面将通过实现上面列举的这些步骤来为地址本插件创建向导。

15.6.1 定义向导扩展

打开插件清单编辑器的“扩展”页，单击“添加”按钮，在弹出的对话框中选择“org.eclipse.ui.newWizards”，单击“完成”按钮将该扩展添加到“扩展”页中。然后，右键单击新加入的扩展，在弹出的菜单中选择“新建”→“wizard”，如图15-11所示。

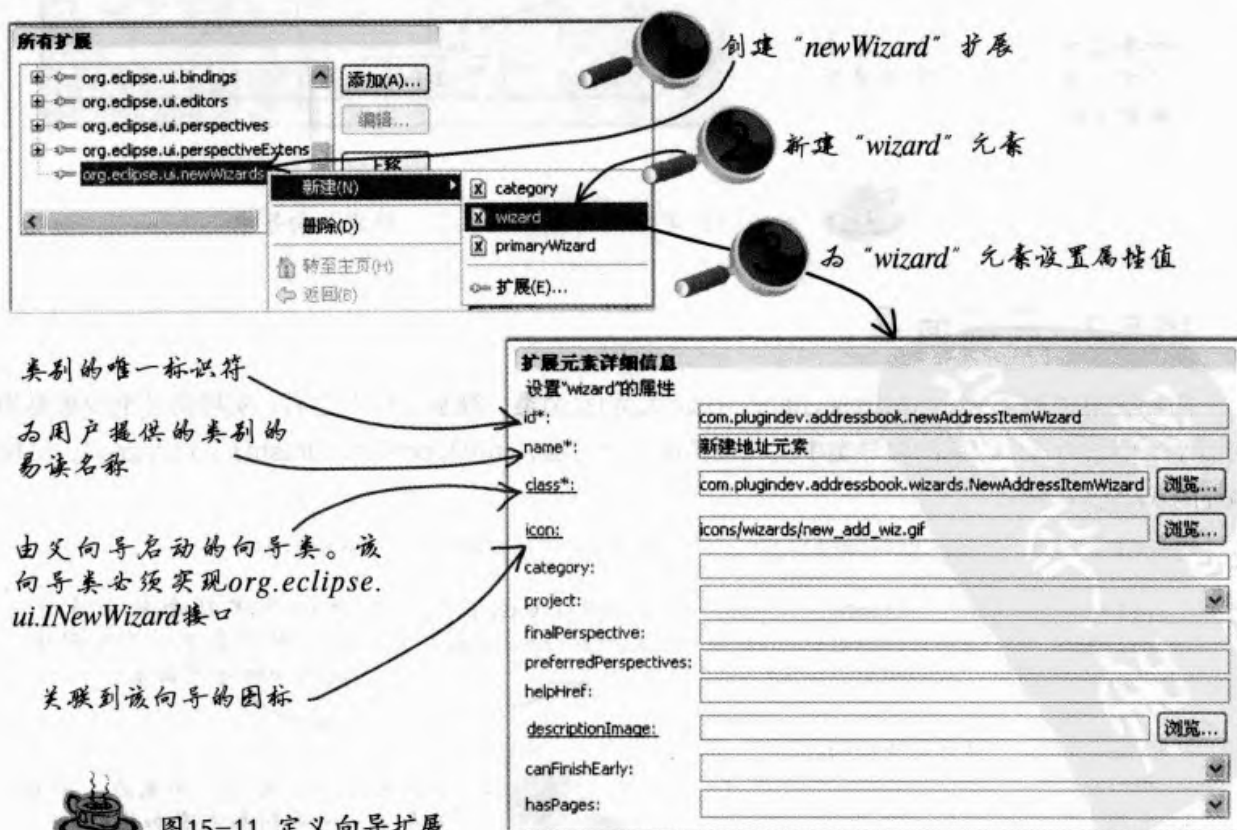


图15-11 定义向导扩展

工作台向导扩展点已经设置了用来启动向导的操作，因而当定义完向导扩展点后，重新启动“地址本”插件，选择“新建”→“其他”，在弹出的向导对话框中的“其他”类别中出现了刚才定义的“新建地址元素”向导，如图15-12所示。

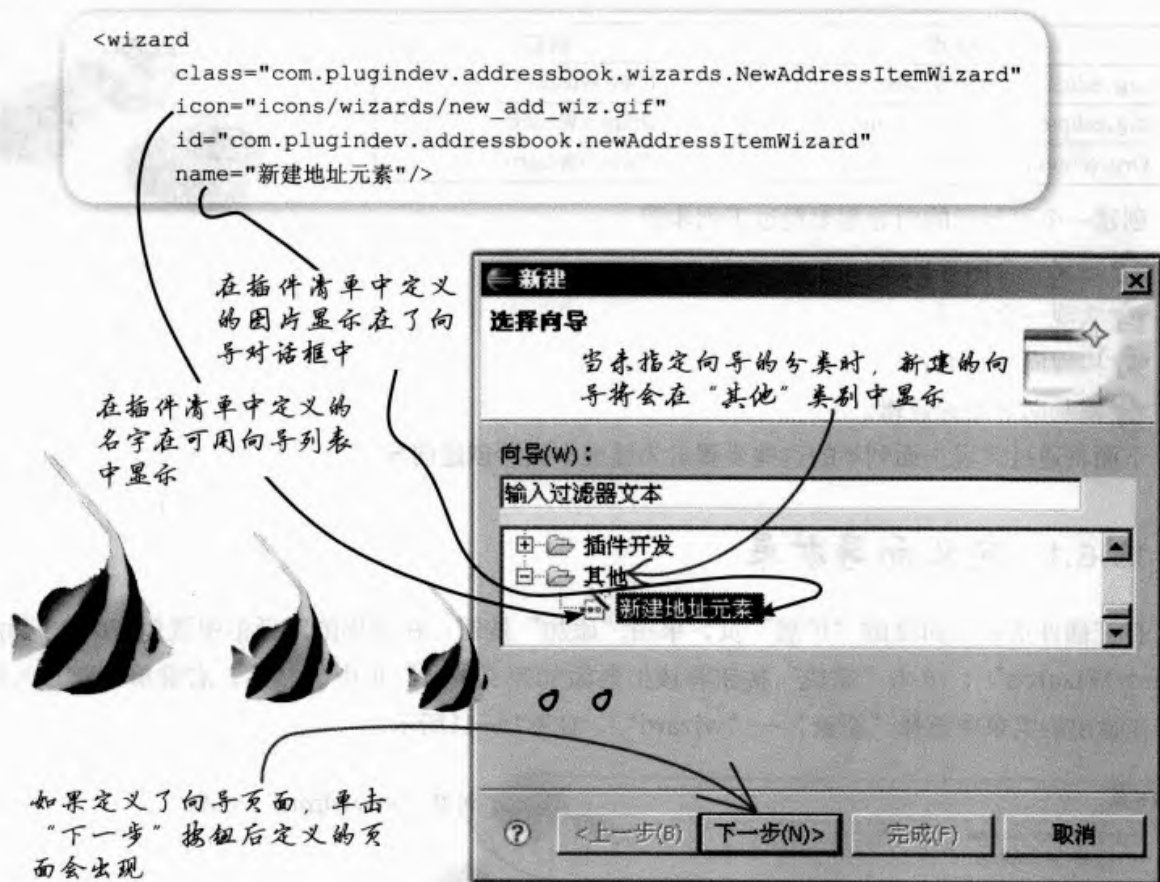


图15-12 定义的向导出现在了“新建”向导中

15.6.2 实现向导

向导类通常都继承org.eclipse.jface.wizard.Wizard类。继承了该类之后，实现向导至少需要为在插件清单编辑器中声明的向导类中实现或扩展三个方法，init(), performFinish(), addPages()。实现向导的代码如下所示。

```
public class NewAddressItemWizard extends Wizard implements INewWizard {
```

```
    private NewAddressItemWizardPage newAddressItemPage;  
    private EditListsConfigWizardPage editListsConfigPage;
```

此向导中包含的向导页面，即将在下一节中创建这两个向导页面类

```
    public void init(IWorkbench workbench, IStructuredSelection selection) {  
    }
```



提供初始化的选择，用来在“新建”向导中的向导列表中添加该向导

```

public void addPages(){
    newAddressItemPage = new NewAddressItemWizardPage();
    addPage(newAddressItemPage);
    editListsConfigPage = new EditListsConfigWizardPage();
    addPage(editListsConfigPage);
}

public boolean performFinish() {
    try{
        getContainer().run(true, true, new IRunnableWithProgress(){
            public void run(IProgressMonitor monitor) throws InvocationTargetException, InterruptedException {
                doFinish(monitor);
            }
        });
    } catch (InvocationTargetException e){
        e.printStackTrace();
        return false;
    } catch (InterruptedException e){
        return false;
    }
    return true;
}

private void doFinish(IProgressMonitor monitor){
    //将会在后续章节实现
}

```

为向导添加向导页面

页面添加的顺序便是页面在向导中出现的顺序

执行“完成”操作

一般在 `performFinish()` 方法中仅创建新的线程和提供异常处理方法，具体的处理操作使用其他的方法来实现

除了默认从“新建”菜单中启动新向导外，还可以从代码中直接启动向导。修改 `com.plugindev.addressbook.actions` 包中的 `AddAddressAction` 类，使该操作实例运行时，打开“新建地址本元素”对话框。使用操作直接启动新向导都需要创建新的向导对话框实例，如下面的代码所示。

```

public void run()
{
    NewAddressItemWizard wizard =
        new NewAddressItemWizard();
    wizard.init(PlatformUI.getWorkbench(), null);
    WizardDialog dialog =
        new WizardDialog(
            PlatformUI.getWorkbench().
            getActiveWorkbenchWindow().
            getShell(), wizard);
    dialog.open();
}

```

该参数为选中的元素，在此处并不依赖被选中元素创建向导，因而将该项设为空

创建向导实例

初始化向导

创建向导对话框

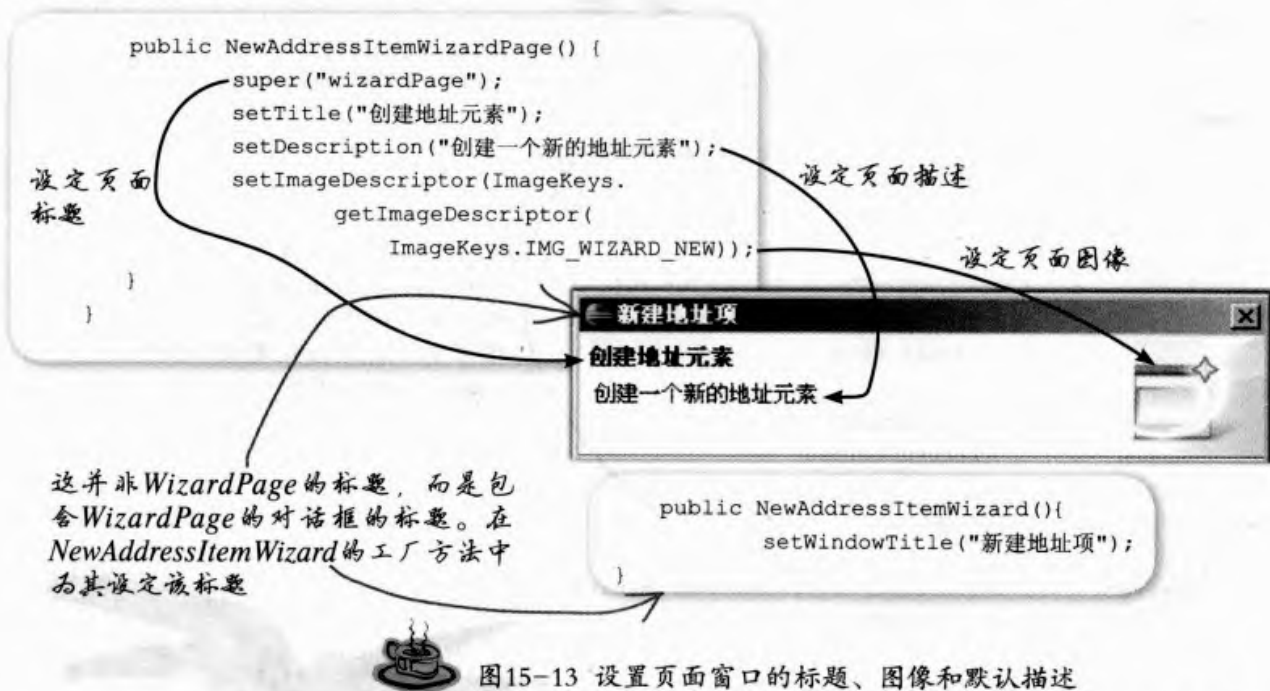
打开向导对话框

15.6.3 实现向导页面

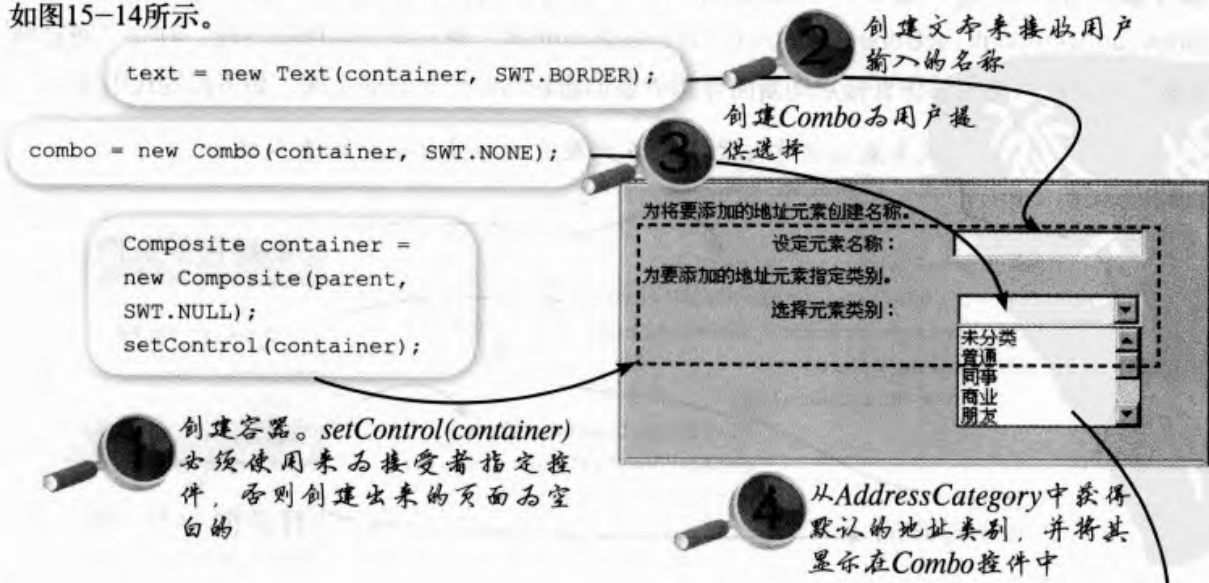
在实现的向导类中定义了两个向导页面，本节将实现这两个向导页面。

NewAddressItemWizardPage用来收集创建地址元素需要的信息。回顾一下第12章创建的AddressItem模型，它包含两个属性，一个是名称，一个是类别。地址本插件默认将地址元素分为了9个类别，可以提供一个下拉按钮允许用户从下拉列表中选择类别。对于名称，只需要简单地为其创建一个文本控件即可。

首先，创建NewAddressItemWizardPage的工厂方法，在该方法中提供一些基本的设置，如页面标题，页面图像等，如图15-13所示。



接下来该创建内容区域了。向导页面使用createControl方法创建内容区域。创建内容区域的过程如图15-14所示。



```
AddressCategory[] allCategories = AddressCategory.getTypes();

for(int i = 0; i < allCategories.length; i++)
{
    String categoryName = allCategories[i].getCategoryName();
    combo.add(categoryName);
}
```



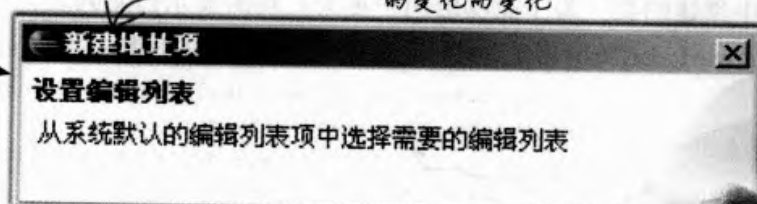
图15-14 设置向导页面的内容

EditListsConfigWizardPage页面放在NewAddressItemWizardPage之后。在地址元素创建之后，该页面用来为用户提供编辑器的“编辑”页中可编辑的“描述信息”列表（有关“编辑”页的更多内容，请参看第13章的相关部分）。示例程序将为该页面提供一个列表选择查看器，支持用户在希望编辑的列表前打勾。

同样，首先为该页面提供初始化设定，如图15-15所示。

```
public EditListsConfigWizardPage(){
    super("选择编辑列表");
    setTitle("设置编辑列表");
    setDescription("从系统默认的编辑列表项中选择需要的编辑列表");
}
```

为EditListsConfigWizardPage
提供标题和默认描述



窗口标题不会随着页面的
变化而变化



图15-15 “设置编辑列表”页面的标题和默认描述

※ 注意 : ※

如果没有为页面设置图像描述符，页面的图像就不会存在了，因此需要为每个页面都指定图像描述符。当希望各页面都使用相同的图像时，有一个便捷的方式。可以在定义的Wizard类的工厂方法中调用setDefaultPageImageDescriptor (ImageDescriptor) 方法来为所有的页面定义默认的图像。如果向导页面也定义了图像，则将会显示该页面定义的图像。




创建完初始化设置之后，需要为EditListsConfigWizardPage提供内容，如图15-16所示。


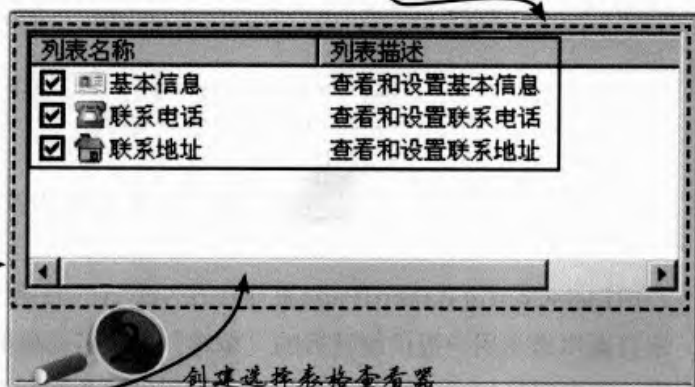
```
final Table table = checkboxTableViewer.getTable();
table.setHeaderVisible(true);
final TableColumn tableColumn = new TableColumn(table, SWT.NONE);
tableColumn.setText("列表名称");
```



```
final TableColumn tableColumn_1 = new TableColumn(table, SWT.NONE);
tableColumn_1.setText("列表描述");
```

 设定表格的列标题

```
Composite container =
    new Composite(parent,
        SWT.NULL);
setControl(container);
```

 为接收者设定控件

 创建选择表格查看器


```
checkboxboxTableView =
    CheckboxTableView.newCheckList(container, SWT.BORDER);
```

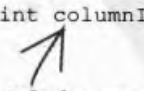


图15-16 设置“编辑列表”页面的内容

CheckboxTableView同其他查看器一样，需要为其设置输入内容提供者和标签提供者。这里使用编辑器的输入SimpleEditorInput作为CheckBoxTableView输入，“编辑页面”的内容提供者MasterContentProvider作为CheckBoxTableView的内容提供者，它们都是前面第14章“编辑器”中创建的类。为了能够给用户更为直观的显示信息列表，示例程序将创建新的标签提供者TableViewLabelProvider。初始化选择列表查看器的过程如图15-17所示。

```
public class TableViewLabelProvider extends LabelProvider
    implements ITableLabelProvider {
        public Image getColumnImage(Object obj, int index) {
            if (index == 0) {
                if (obj instanceof BasicAddressList) {
                    return ImageCache.getInstance().getImage(
                        ImageKeys.getImageDescriptor(ImageKeys.IMG_SCROL_BASIC));
                }
            }
            //省略了部分相似代码
            return null;
        }
        public String getColumnText(Object element, int columnIndex) {
```

 设置列表图像，地址本插件默认提供了三个信息列表：基本、电话、区域，需要分别为这三个列表设置图像

 设置列表显示的内容。当为第一列时 (columnIndex == 0)，显示列表名称，当为第二列时，显示列表的描述

```
if (element instanceof AddressList) {
    AddressList addressList = (AddressList) element;
    switch (columnIndex) {
```

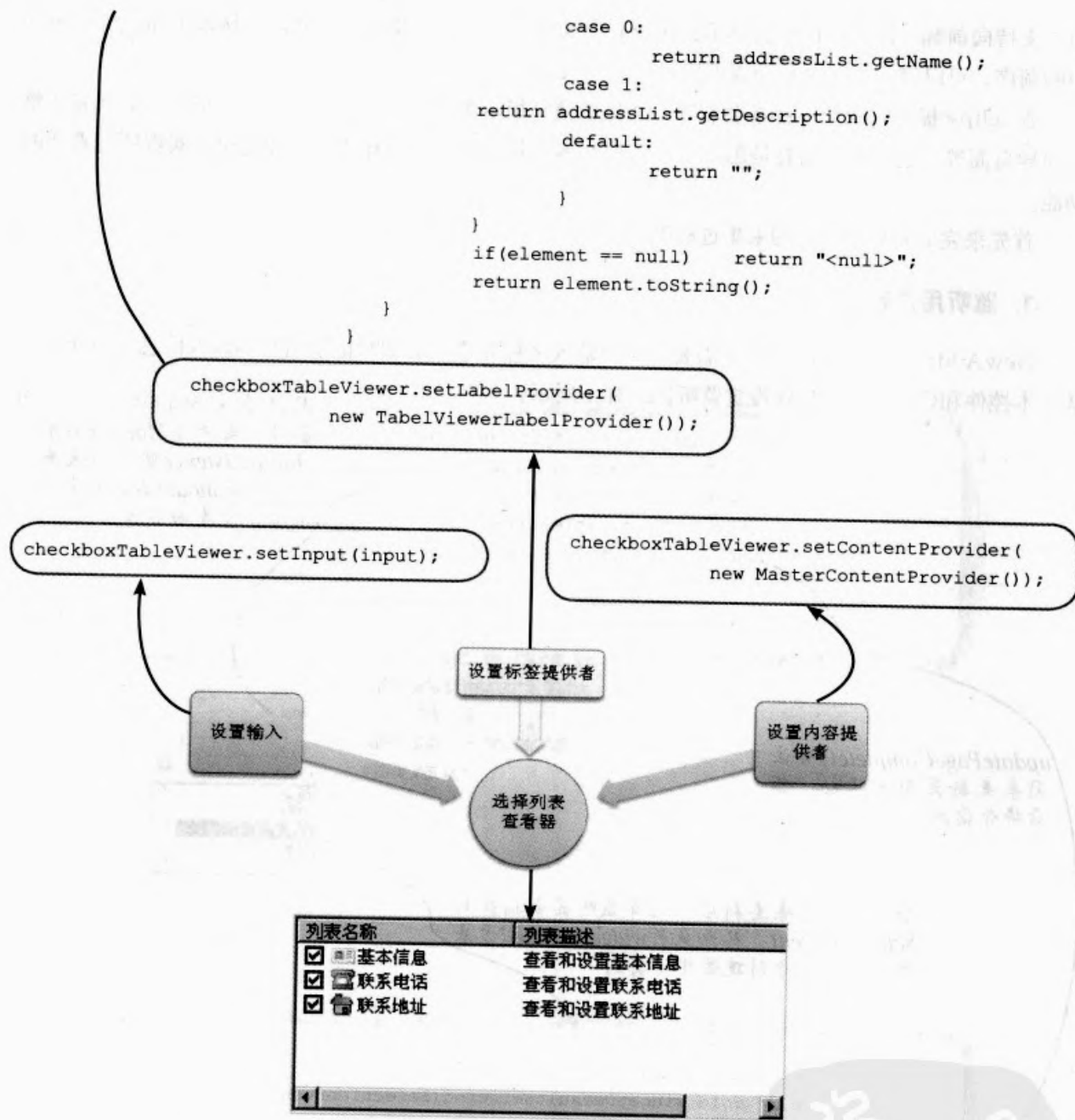


图15-17 初始化选择列表查看器

※ 注意 ※

`checkboxTableView.setInput(input)`方法不是在`createPartControl()`中定义的，稍后将会介绍如何设置查看器的输入。



15.6.4 添加向导处理逻辑

当多个页面被添加到向导中时，从向导架构继承而来的控制逻辑就决定是否激活“下一步”按

钮。支持向前翻页的逻辑已经被继承到向导处理架构中了，不需要再进行处理。按照页面添加到向导中的顺序，可以进入下一个页面或返回到前面的页面。

在Eclipse提供的向导中经常会看到一些提示信息指示用户应该如何输入有效的格式，如何完成整个向导页面等。这些提示信息是怎么实现的呢？本节将为创建的向导添加处理逻辑，实现所有希望的功能。

首先来完成对用户输入的采集过程吧！

1. 监听用户输入

NewAddressItemWizardPage需要在用户输入名称和选择类别时记录用户的输入信息，因此需要为文本控件和ComboBox控件设定监听器，如下所示。

```
text.addModifyListener(new ModifyListener(){
    public void modifyText(ModifyEvent e) {
        Text text = (Text)e.getSource();
        name = text.getText();
        updatePageComplete();
    }
});
```

用户在文本控件中输入内容时，会产生ModifyEvent，ModifyListener监听到该事件后会执行modifyText方法，记录用户修改的名称。

updatePageComplete()方法用来更新页面的状态，稍后将介绍此方法。

为将要添加的地址元素创建名称。

设定元素名称：TBeg

为要添加的地址元素指定类别。

选择元素类别：

朋友
家庭
VIP
师长
伴侣

当选择了元素类别后，选择监听器会捕获到SelectionEvent，从而执行widgetSelected方法根据类别名称创建选择的类别。

```
combo.addSelectionListener(new SelectionListener(){
    public void widgetDefaultSelected(SelectionEvent e) {}
    public void widgetSelected(SelectionEvent e) {
        Combo combo = (Combo)e.getSource();
        String str = combo.getText();
        selectedCategory = AddressCategory.getCategoryByName(str);
        updatePageComplete();
    }
});
```

2. 对用户的输入进行反馈

向导页面标题下面的消息区域可以动态地显示反馈内容。通常，该区域用来指示用户在继续下一个向导页面或执行操作之前，还需要输入其他的信息。这里创建了updatePageComplete()方法支持

对输入内容的反馈。地址本插件将在确定了初始内容时，调用updatePageComplete()方法一次，然后在内容更改的时候，各个文本框监听器将再次调用该方法；该方法检查当前文本框和ComboBox的内容，当内容不符合要求时，显示一个错误或警告消息，并适当的启用或禁用“下一步”和“完成”按钮。

图15-18至图15-21演示了实现对用户输入反馈的步骤。

设置页面未完成，这样，“下一步”和“完成”按钮就被禁用

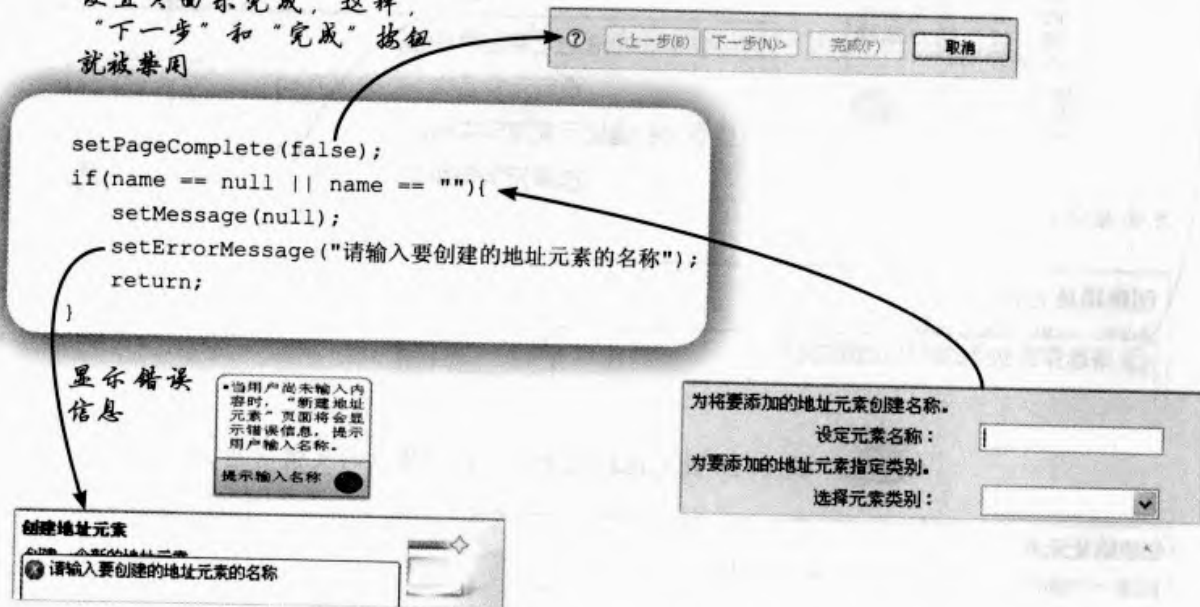


图15-18 设定地址元素名称为空时的反馈消息

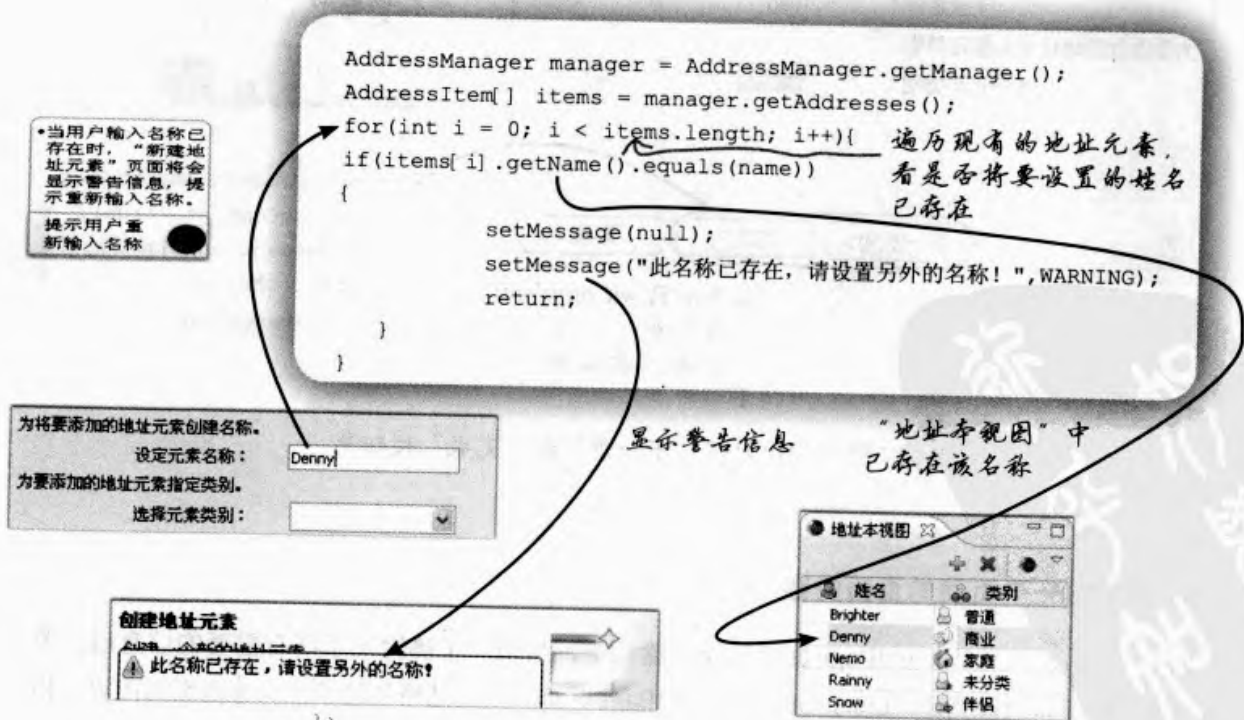


图15-19 设定地址元素名称已存在时的反馈消息


```
if(selectedCategory == null){  
    setMessage(null);  
    setErrorMessage("请选择要创建的地址元素的类别");  
    return;  
}
```

•当已输入名称,但还未选择元素类别时,会出现错误消息,提示用户选择元素类别。

提示用户输入元素类别

显示错误信息

为将要添加的地址元素创建名称。

设定元素名称:

Tbeg

为要添加的地址元素指定类别。

选择元素类别:

创建地址元素

创建一个新的地址元素

请选择要创建的地址元素的类别



图15-20 设定地址元素类别未选择时的反馈消息

创建地址元素

创建一个新的地址元素

提示消息消失

为将要添加的地址元素创建名称。

设定元素名称:

Tbeg

为要添加的地址元素指定类别。

选择元素类别:

未分类

•当元素类别选择后,所需要收集的信息已经完成,这时可以启动“下一步”和“完成”按钮了。

```
setPageComplete(true);  
setMessage(null);  
setErrorMessage(null);
```

当为setPageComplete()设定为真值时,“下一步”和“完成”按钮就已经可用了



图15-21 启用“下一步”和“完成”按钮

3. 基于前一个页面的页面内容

NewAddressItemWizardPage页面已经构建完毕,当用户正确输入了所有需要的信息后,单击“下一步”按钮就会跳到EditListsConfigWizardPage页面。虽然已经创建了该向导页面的内容,但是该页面中的内容不会更新。Eclipse的向导架构提供了setVisible()方法,插件开发者需要覆写该方法,在每次页面变为可见时更新它的内容。如图15-22所示。

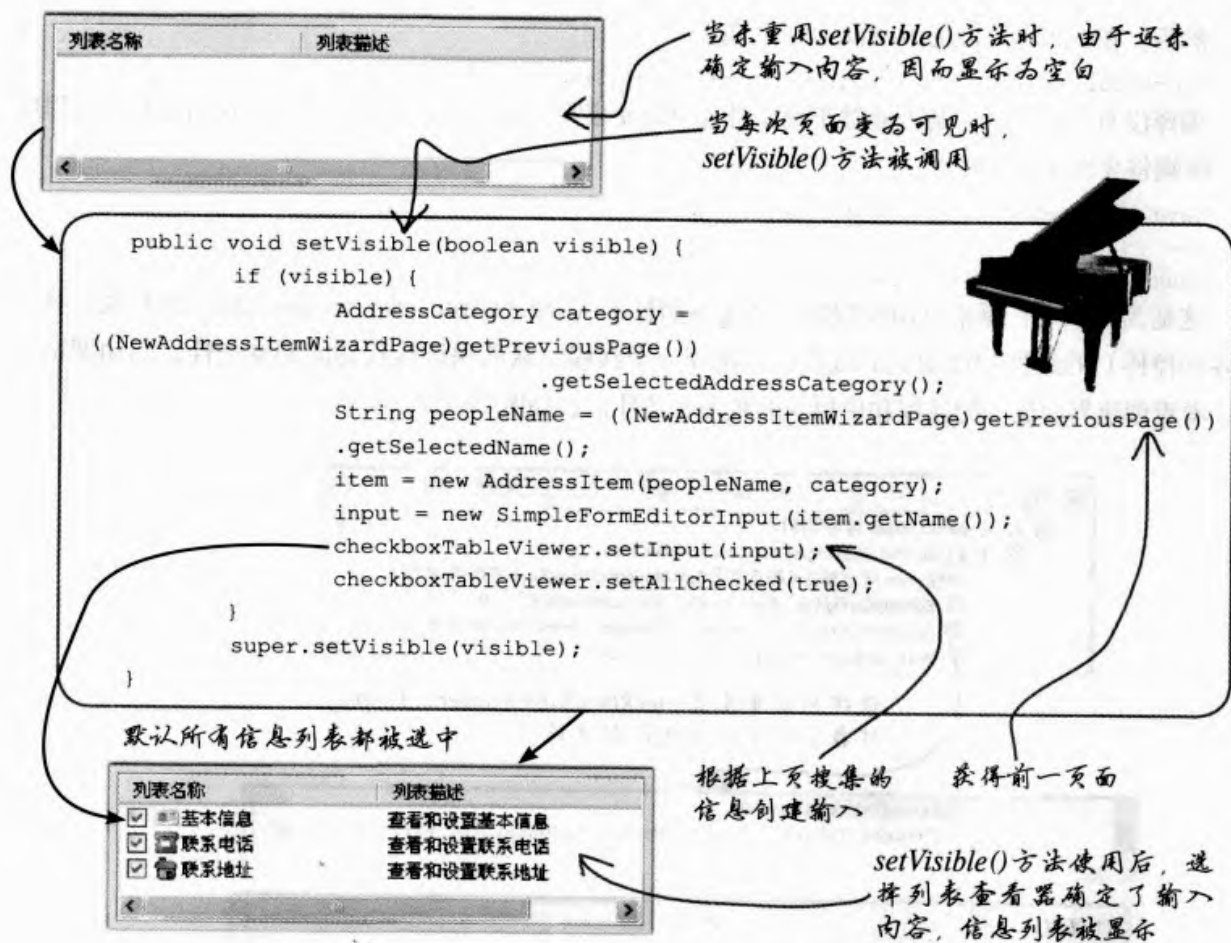


图15-22 根据上一个页面收集的信息更新页面内容

4. 完成向导

收集完信息后，就可以根据收集到的信息进行处理了，这要通过填充NewAddressItemWizard类中的doFinish()方法来实现。

首先在EditListsConfigWizardPage类中创建获得选中的列表的方法和一些存取方法，代码如下所示。

```

public AddressList[] getSelection() {
    checked = checkboxTableViewer.getCheckedElements();
    int count = checked.length;
    AddressList[] extracted = new AddressList[ count];
    System.arraycopy(checked, 0, extracted, 0, count);
    return extracted;
}

public AddressItem getAddressItem(){
    return item;
}

public SimpleFormEditorInput getEditorInput(){
    return input;
}

```

获得选中的元素列表数组

复制选中的元素列表数组

获得新创建的地址元素，用来在单击“完成”按钮后在模型中添加地址元素

获得同用户设定相匹配的编辑器输入，用来在单击“完成”按钮时打开编辑器

然后，在doFinish()方法中加入如下代码。

```
AddressList[] lists = editListsConfigPage.getSelection();
```

编译没有错误，那么是否能够正确获得选中的元素呢？运行地址本插件后，发现当运行该程序时，终端将会出现如下错误。

```
java.lang.reflect.InvocationTargetException
```

```
.....
```

```
Caused by: org.eclipse.swt.SWTException: Invalid thread access
```

这是为什么呢？原来Eclipse规定了创建界面显示（org.eclipse.swt.widget.Display对象和其上的各种控件）的线程和使用它的线程必须是同一个线程，从而保证线程访问的安全性。启动调试状态，看看创建界面显示的线程和使用它的线程各是什么。如图15-23所示。

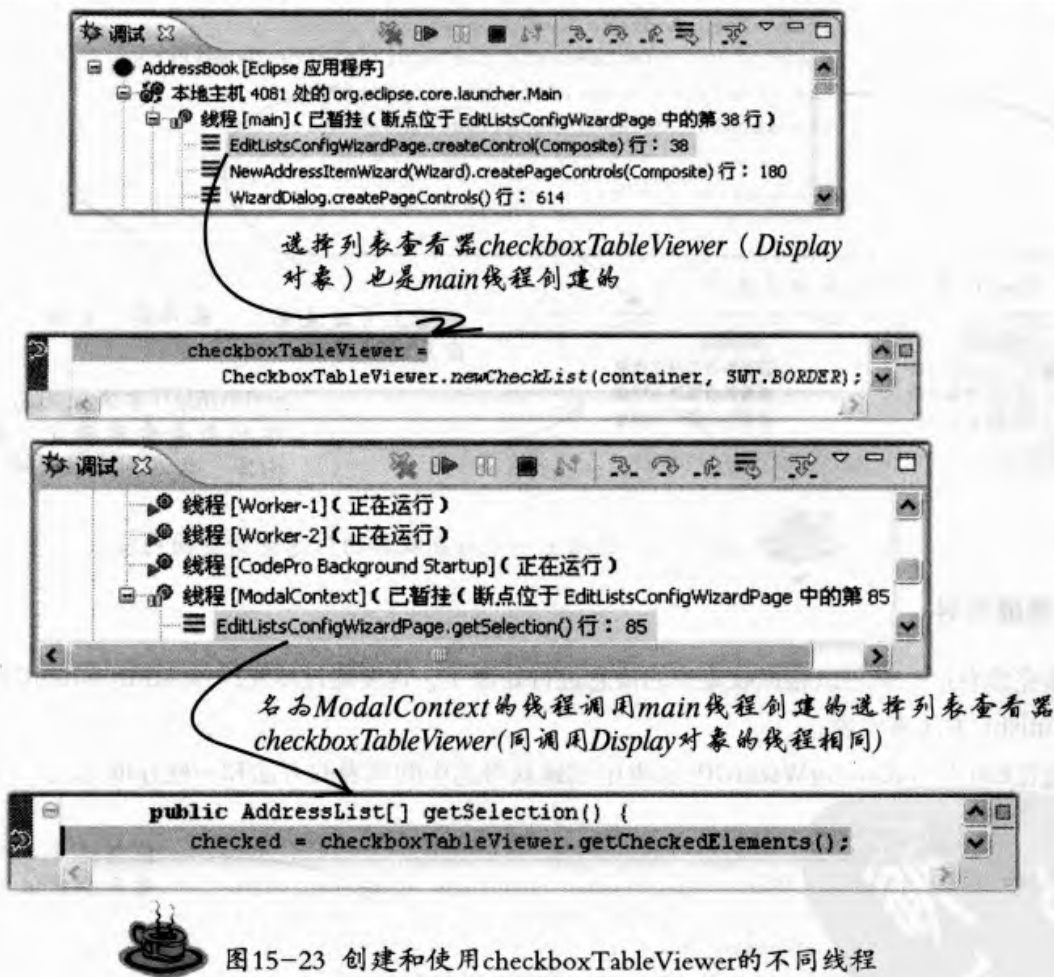


图15-23 创建和使用checkboxTableView的不同线程

为了避免上述现象的发生，处理类似的问题时需要使用Display对象提供的asyncExec()方法。asyncExec()方法是外部线程调用本地线程的桥接方法，这样创建Display对象的线程和使用Display对象的线程都会是main()线程了。

下面来完成doFinish()方法。先来看看doFinish()方法的主体，代码如下所示。

```
private void doFinish(IProgressMonitor monitor){  
    Display display = null;
```

```

Shell shell = getShell();
if (shell == null) display = PlatformUI.getWorkbench().getDisplay();
else
display = shell.getDisplay();
display.asyncExec(new Runnable(){
    public void run() {
        //.....
    }
});

```

获得display对象

在此处的处理由创建display的线程执行，在这里加入获得页面收集到的信息，更新模型的方法

在run()方法中包含的是具体的处理信息的过程，如图15-24所示。



图15-24 创建和使用checkboxTableViewer的不同线程

至此就全部完成了新向导的创建步骤。

※ 注意: ※



*Display*对象提供了`syncExec()`和`asyncExec()`两个外部线程调用`display`本地线程的桥接方法，这两个方法的区别在于`syncExec()`方法是同步方法，当本地还未处理完时，外部线程不会继续执行；而`asyncExec()`方法是异步方法，不论本地线程有没有处理完，外部线程都会继续执行。

15.7 本章小结

本章介绍了在Eclipse插件开发中如何创建对话框和向导。Eclipse本身提供了许多有用的对话框，通过重用这些作为Eclipse Platform一部分的对话框，插件开发者可以全面地增强插件的功能，提升插件的用户界面集成程度，同时帮助用户以一致的方式执行所需的任务，并且提高插件开发效率。开发者也可以在满足需求的内置对话框或向导时，按照Eclipse对话框和向导的架构创建自己的对话框和向导，来支持希望让用户执行的任务。

添加对话框和向导的过程都既可以通过扩展的方式来完成，也可以通过在插件定制逻辑中直接在代码中调用相应的类来完成。

下一章将介绍如何使插件为“首选项”页面添加内容。

可以休息一会
了，放松一下



第16章 首选项 (Preferences)

拉开崭新的学习帷幕

从第12章开始，本书介绍了Eclipse的三大部件：视图、编辑器和向导，在创建完它们之后，开发的插件可能需要这些部件支持用户改变某些默认设置，以此来增强插件满足用户需求的能力。Eclipse提供了达到此目的的手段——“首选项”。本章通过添加针对地址本插件的首选项页面，讨论如何使用Eclipse的首选项，包括创建Eclipse首选项页面和用来记录和恢复插件首选项的技术。

本章内容包括：

- ★讨论首选项的页面结构
- ★使用示例模板为插件添加示例首选项页面
- ★介绍功能强大的首选项字段编辑器
- ★创建首选项页面来改变视图、编辑器的初始化设定

进入第16章



大多数Eclipse插件都提供了用户可配置的首选项，来控制插件执行和显示信息的方式。使用首选项页面，可以定义设置(setting)，用户使用这些设置来控制插件的行为。这些设置作为与插件相关的数据被保存起来，这些数据可以用于当前工作空间中。

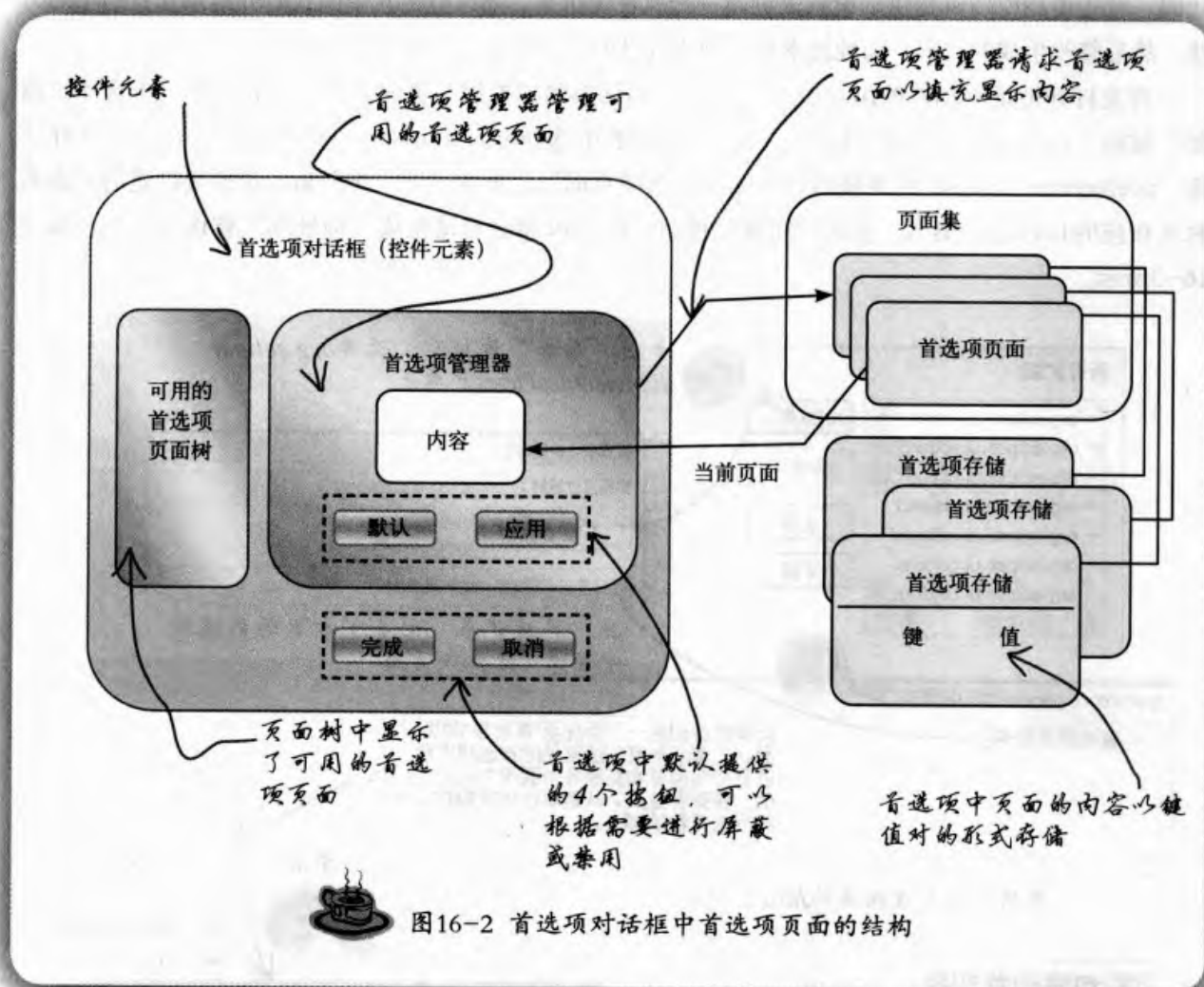
首选项页虽然功能很强大，但是它并不用来配置当前工作空间的资源，而是用来为编辑器、视图或其他在资源上执行的同用户交互不频繁的操作服务。首选项值与单独的资源无关，而通常与活动的工作空间相关。在资源实例中存储的数据不适用于使用首选项来配置，可以通过属性视图、属性对话框等完成。

如在“地址本插件”中，关于具体元素的AddressItem, AddressList等对象属性值的设定不适合使用首选项来设置，而视图显示的行列数、编辑器显示的页面数、编辑器的布局可以通过首选项页面来配置，如图16-1所示。



16.1 首选项页面结构

图16-2描述了位于通用“首选项”对话框中的首选项页面的结构。



由图16-2可以看到, 首选项对话框有两个主要的部件, 一个是首选项页面树, 用来以树形显示可用的首选项页面, 另一个是首选项管理器, 它管理首选项页面的添加、修改和删除。在构建每个首选项页面时, 都会有一个Composite对象, 可以在其上增加其他SWT/JFace部件。当在页面树中选中该首选项页面时, 这个Composite对象可以得到焦点。

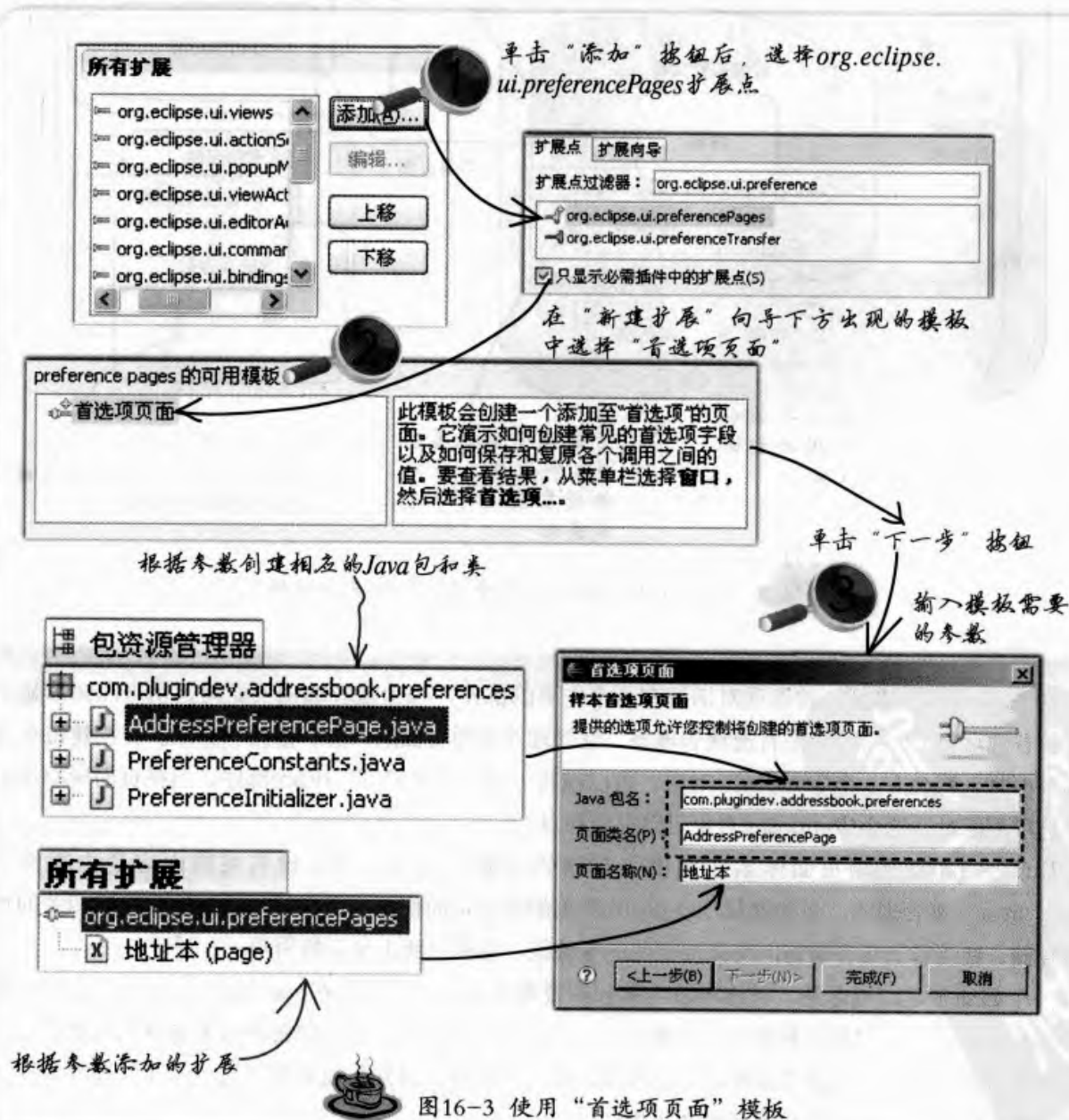
Eclipse的首选项页面体系还提供了存储的功能, 同首选项存储有关的内容是由插件类 (Activator) 来管理的。包含底层Composite对象的活动页面使用它所在插件的插件类找到相关的首选项存储, 使用首选项存储访问与插件相关的存储值, 并在页面上显示给用户。

一个必须牢记的概念是, 首选项值与单独的资源无关。它们保存在org.eclipse.core.runtime插件的state目录中, 它们并没有保存在资源库中。可以通过定义默认首选项键的值来提供初始化的工作空间状态, 并且当某个首选项设置的值更改时, 使用监听器来通知其他对象。也可以通过读取或导出首选项值, 以便将首选项在工作空间之外共享。

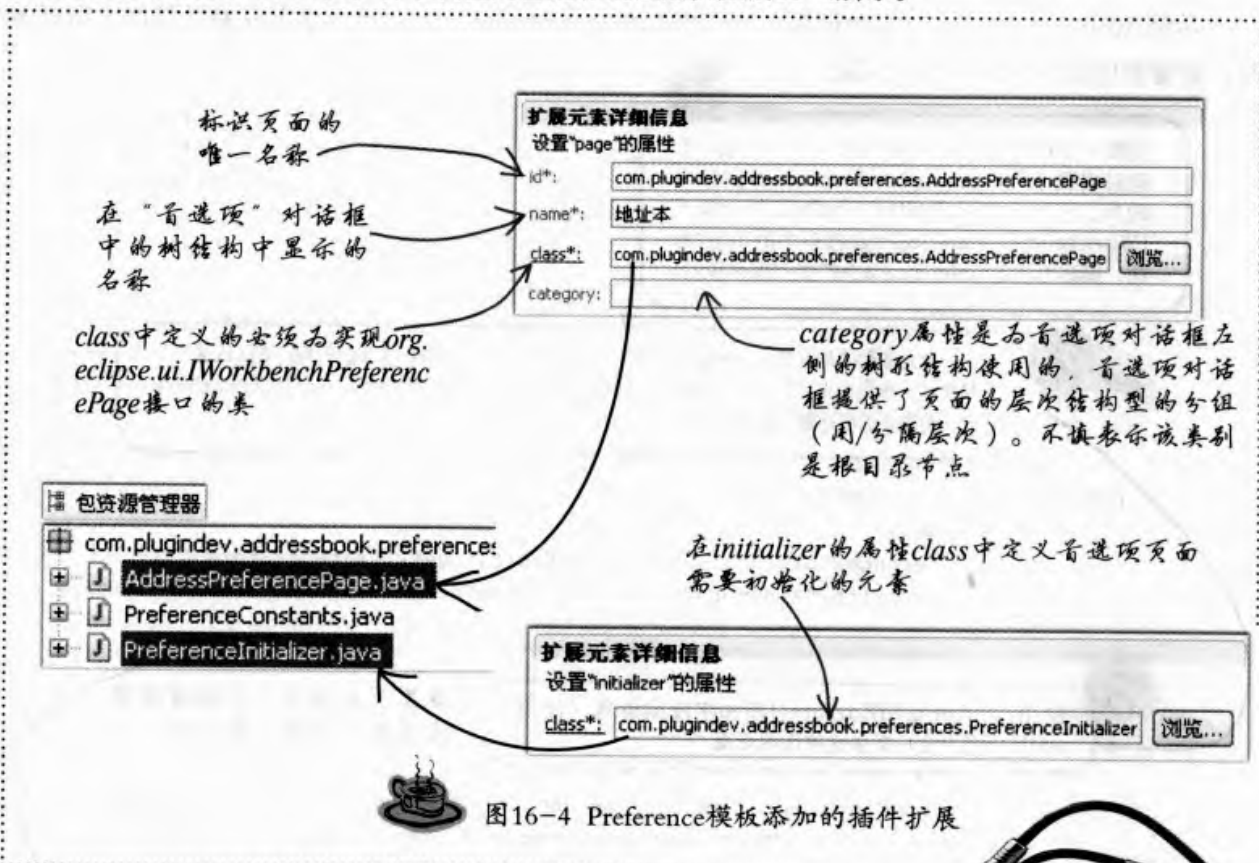
16.2 添加首选项页面

添加首选项页面也需要在插件清单中添加扩展项 (org.eclipse.preferencePages) 并创建相应的代码。首先使用Eclipse提供的模板来创建一个属性编辑页, 通过分析其中的代码介绍添加首选项的方法, 然后修改生成的代码, 为地址本插件创建自己的“地址本”首选项。

首先打开地址本插件的plugin.xml文件, 并切换到“扩展”页面。在“所有扩展”中选择“添加”按钮, 打开“新建扩展”向导, 从扩展点列表中选择org.eclipse.ui.preferencePages, 选择下面“preference pages的可用模板”中的“首选项页面”, 单击“下一步”后, 选择要创建的页面名称及相应的Java包和类名, 单击“完成”按钮, Eclipse将会自动生成“地址本”首选项页面, 如图16-3所示。



由模板生成代码后，打开扩展页，发现扩展页中不仅增加了org.eclipse.ui.preferencePages扩展，还增加了名为org.eclipse.core.runtime.preferences的扩展点。preferences扩展点允许插件为Eclipse首选项机制添加新的首选项作用域（scope，本书中不涉及作用域的概念），并且指定在运行时初始化默认首选项中的值。各个属性的含义和设定的值如图16-4所示。



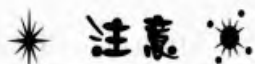
16.3 示例首选项

Preference模板生成了一些代码，本节将通过分析这些代码，介绍生成“首选项”页面的过程。

16.3.1 示例首选项页面

首选项页面必须实现org.eclipse.ui.IWorkbenchPreferencePage接口，开发者可以自己实现这个接口中用到的方法，但通常情况下均通过扩展抽象类org.eclipse.jface.preference.PreferencePage来简化处理过程。Preference类提供了实现用户界面，获得首选项设置，以及保存设置值的接口和方法。如果开发者需要的是文本编辑、选择框编辑等简单的编辑部件，可以扩展其子类FieldEditorPreferencePage来进一步简化实现过程。

FieldEditorPreferencePage同org.eclipse.jface.preference包中的字段编辑器类一起，提供了一种表示和捕获简单首选项的快速简便的方式。



注意

*FieldEditorPreferencePage*类功能很强大，它会处理加载、验证和保存字段编辑器内容的大多数工作。但它假定页面上的所有首选项都是字段编辑器，当需要创建更为复杂的首选项编辑页面时，仍然需要使用*PreferencePage*类。

示例*AddressPreferencePage*扩展了*FieldEditorPreferencePage*，当构建字段编辑器首选项页面时，需要实现如图16.5所示的步骤。

实现一个扩展了*FieldEditorPreferencePage*类同时实现了*IWorkbenchPreferencePage*接口的类。*FieldEditorPreferencePage*并未实现*IWorkbenchPreferencePage*接口中的*init()*方法，而该方法是*PreferencePage*扩展点所必须的

```
public class AddressPreferencePage
    extends FieldEditorPreferencePage
    implements IWorkbenchPreferencePage
{
    //.....
}
```

实现一个无参数的构造方法。在这个方法中，为*FieldEditorPreferencePage*的构造方法传递一个布局样式，布局样式可以从GRID和FLAT中选择。并且为首选项页面指定一个*PreferenceStore*对象，关联首选项存储和首选项页面

```
public AddressPreferencePage() {
    super(GRID);
    setPreferenceStore(Activator.getDefault().getPreferenceStore());
    setDescription("A demonstration of a preference page implementation");
}
```

使用*createFieldEditors()*添加字段编辑器（使用*addField()*方法）来创建首选项页面

设置页面描述。页面描述将会出现在页面标题下方

```
addField(new DirectoryFieldEditor(PreferenceConstants.P_PATH,
    "&Directory preference:", getFieldEditorParent()));
```

```
addField(
    new BooleanFieldEditor(
        PreferenceConstants.P_BOOLEAN,
        "&An example of a boolean preference",
        getFieldEditorParent()));
```

创建一个目录字段编辑器

```
addField(new RadioGroupFieldEditor(
    PreferenceConstants.P_CHOICE,
    "An example of a multiple-choice preference",
    1,
    new String[][] { { "&Choice 1", "choice1" }, {
        "C&hoice 2", "choice2" }
    }, getFieldEditorParent()));
```

创建一个布尔值字段编辑器

```
addField(new StringFieldEditor(
    PreferenceConstants.P_STRING,
    "A &text preference:",
    getFieldEditorParent()));
```

创建一个选择框字段编辑器

创建一个字符串字段编辑器

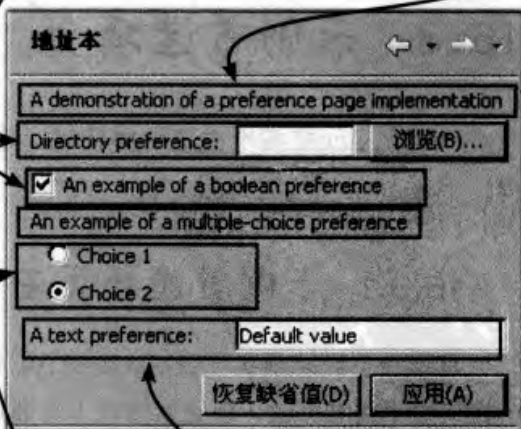


图16-5 示例代码实现字段编辑器的步骤

如果是扩展PreferencePage类, 要做更多的事情, 如下所示。

★为了实现用户界面, 需要定制createContents() (而不是createFieldEditors()) 方法来创建页面所需的适合的控件。

★定制performOk方法来保存首选项值。

★定制performDefault方法来应用首选项定义的默认值。

图16-6通过类图介绍了同定制首选项页面相关的类及一些有用的方法。

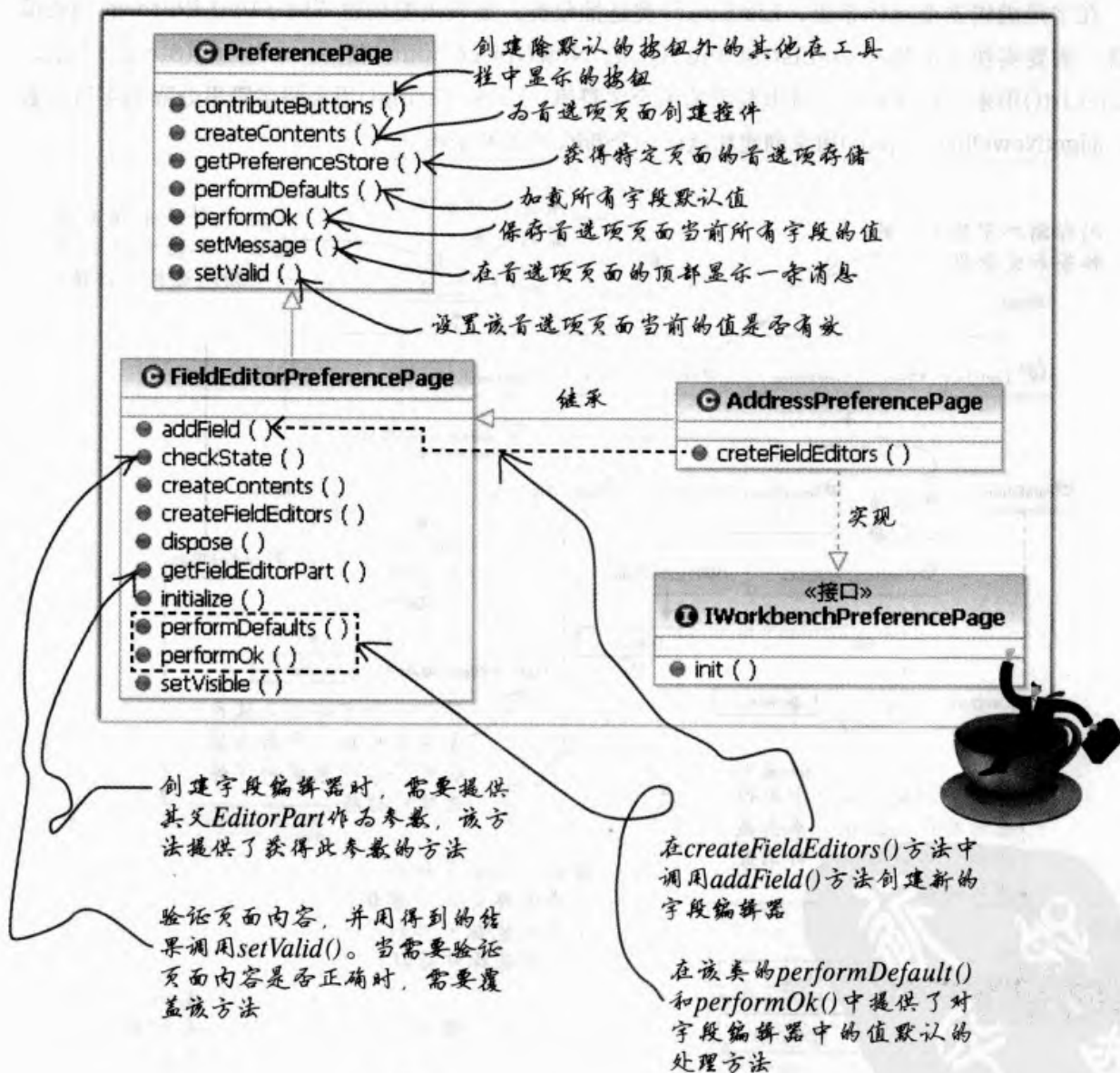


图16-6 同定制首选项页面相关的类及方法

16.3.2 字段编辑器

向页面添加的字段编辑器提供了关键部分的处理支持。org.eclipse.jface.preference提供了很多

种不同的字段编辑器，每个字段编辑器包含了一个或多个SWT控件。抽象类FieldEditor作为字段编辑器共同的超类，为FieldEditorPreferencePage提供了通用的接口，使字段编辑器页面可以在不需要了解具体的字段编辑器类型的情况下访问它们。

字段编辑器用来显示和处理简单类型的首选项，如字符串、整数、颜色、字体等。如果首选项用于配置这些简单的值，就不再需要编写烦琐的处理逻辑，直接可以对这些首选项进行加载、显示、验证和保存等操作。图16-7显示了Eclipse平台提供的字段编辑器的继承体系，并介绍了它们的用途。

在字段编辑器继承体系里，ListEditor类是抽象类，如果需要创建不同于PathEditor的列表编辑器，需要实现其中的createList(String []),getNewObjectInput()和parserString(String)方法，createList()用来为合并列表字符串数组为单个字符串，parserString()用来将字符串分隔为字符串数组，而getNewObjectInput()用来创建和返回一个新的列表字符串。

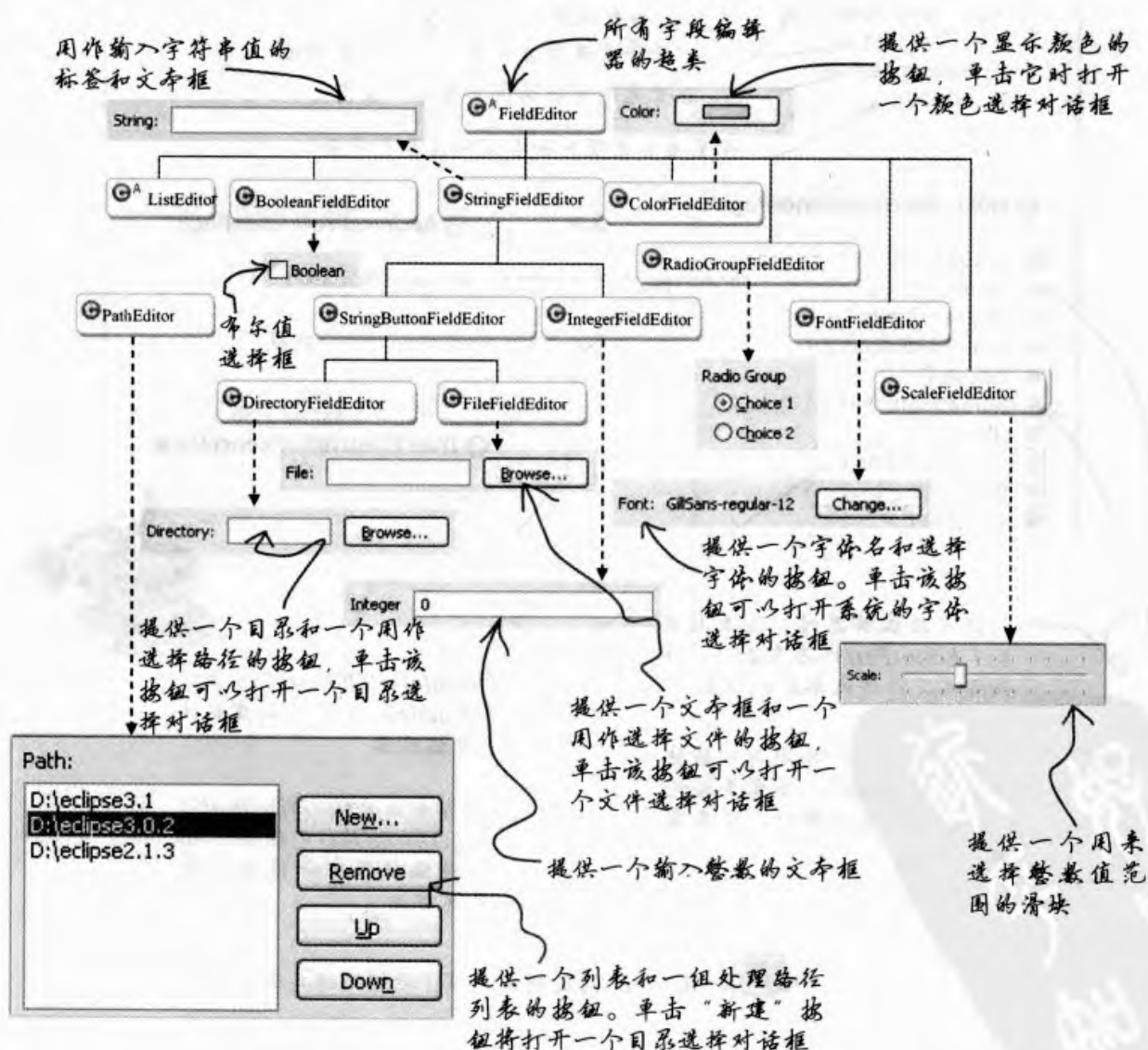


图16-7 字段编辑器的继承体系

各个字段编辑器自己可以保存页面的设置值，示例源代码还提供了默认初始化例程来为各个字段编辑器提供默认的首选项值。

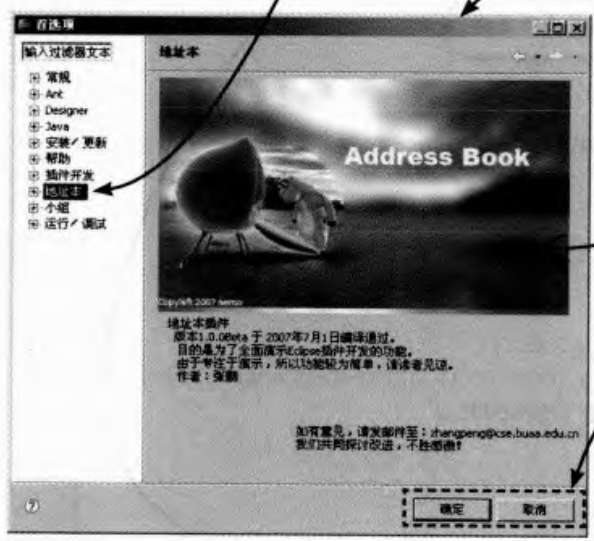
16.4 为例子创建首选项页面

插件创建的首选项的一个常用的方式是，根首选项页面介绍有关产品的基本信息，而子首选项页面包含实际的首选项。本节将按照这样的方式创建首选项页面，为地址本插件创建一个根首选项页面和分别针对“地址本视图”和“地址本编辑器”的两个叶首选项页面。

16.4.1 创建根首选项页面

根首选项页面很简单，AddressRootPreferencePage直接继承了PreferencePage类，并且只在createContents()方法中提供了一些同地址本插件有关的描述信息。需要注意的是，示例程序在根首选项页面中去掉了“恢复缺省值”和“应用”两个按钮，该页面没有可供编辑的项，因而不需要使用这两个按钮，需要调用noDefaultAndApplyButton()方法来去掉这两个按钮。创建完根首选项页面后，修改plugin.xml中的代码，将同org.eclipse.jface.preferencepages扩展点中定义的页面中的class属性对应到创建的AddressRootPreferencePage类上。如图16-8所示。

```
<page
    class="com.plugindev.addressbook.preferences.AddressRootPreferencePage"
    id="com.plugindev.addressbook.preferences.AddressPreferencePage"
    name="地址本"/>
</extension>
```



当插件作为产品发布时，一般在首选项页面的根节点放入该产品的Logo和一些描述信息，典型的如SWT Designer。如果最终生成的是RCP程序，则可以在帮助菜单中的“关于...”中加入

在init(workbench)方法中调用noDefaultAndApplyButton()方法，取消这两个按钮的默认显示

```
public void init(IWorkbench workbench) {
    // Initialize the preference page
    noDefaultAndApplyButton();
}
```



图16-8 地址本插件首选项根页面

16.4.2 创建“视图”子首选项页面

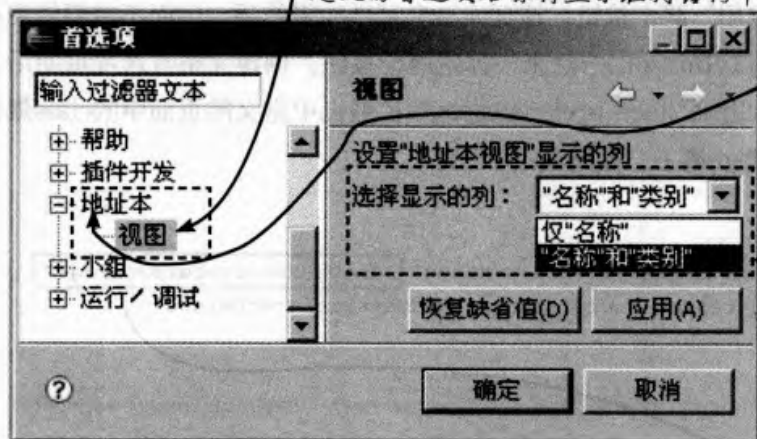
Eclipse为插件开发者提供了首选项页面按层次分组的机制, 允许为首选项页面添加子首选项页面。这需要通过在插件清单中添加相对路径来获得。本节将为“地址本”首选项添加“视图”子首选项页面, 使得可以在首选项页面中定制“地址本”视图显示列的个数。

创建“视图”首选项的过程如图16-9所示。

在插件清单中的org.eclipse.ui.preferencePages扩展点中增加一个新的首选项页面, 在plugin.xml中的代码如下所示

```
<page
    category="com.plugindev.addressbook.preferences.AddressPreferencePage"
    class="com.plugindev.addressbook.preferences.ViewPrefPage"
    id="com.plugindev.addressbook.preferences.ViewPrefPage"
    name="视图"/>
```

定义的首选项名称将显示在树结构中



此处为“地址本”根首选项页面的标识, 放在该category下, 表示该首选项页面应为“地址本”根首选项的子页面。如果有多层结构, 则每层用“/”分隔

创建一个ComboFieldEditor字段编辑器, 该字段编辑器是在SWTDesigner生成的com.swtdesigner.preference.*包中定义的

创建清单声明的ViewPrefPage类, 继承字段编辑器首选项页面

```
public class ViewPrefPage extends FieldEditorPreferencePage
implements IWorkbenchPreferencePage {

    private ComboFieldEditor comboFieldEditor;

    public ViewPrefPage() {
        super(FieldEditorPreferencePage.GRID);
        setPreferenceStore(Activator.getDefault().getPreferenceStore());
        setDescription("设置\“地址本视图\“显示的列");
    }

    protected void createFieldEditors() {
        {
            comboFieldEditor = new ComboFieldEditor(
```

```

getFieldEditorParent();

addField(comboFieldEditor);
    }
    public void init(IWorkbench workbench) {
    }
}
    
```

PreferenceConstants.P VIEW COMBO,
"选择显示的列:",
new String[][] {{ "仅\名称", "", "0"},
{ "\名称"和"类别", "", "1" } },

此处的String[][]是二维数组，第一维表示“名称”，即用户看到的名称，第二维表示的是“值”，即该选择实际表示的值

Combo字段编辑器中
标签的显示名称

在PreferenceConstants中声明首选项常量，作为Combo字段的标识

public static final String P_VIEW_COMBO = "comboPreference";

在PreferenceInitializer中设置默认的“视图”首选项值

store.setDefault(PreferenceConstants.P_VIEW_COMBO, "1");

默认首选项值为“1”，表示默认情况下“名称”和“类别”两列都显示

图16-9 创建视图首选项的过程

创建完“视图”首选项后，需要将其挂接到“地址本”视图，使地址本视图可以响应首选项设置的更改。这通过在地地址本视图类中声明一个属性监听器来实现，在监听到“地址本”首选项页面的变化后，更新地址本视图的显示，如图16-10所示。

在AddressView类中声明属性监听器

```

private final IPropertyChangeListener
propertyChangeListener = new IPropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent event) {
        if (event.getProperty().equals(
            PreferenceConstants.P_VIEW_COMBO))
            updateColumnNumbers();
    }
}
    
```

监听到首选项值的更改

响应首选项更改，更新表格列的数量

设置“地址本视图”显示的列
选择显示的列: 仅名称
恢复缺省值(D) 应用(A)

初始状态显示为两列

在“视图”首选项页面中设置为仅显示名称

姓名	类别
Engler	音乐
Denny	商业
Nemo	家庭
Rainny	未分类
Snow	伴侣
Tbag	未分类

根据“视图”首选项页面的首选项值更新视图的表格列

将声明的属性监听器注册到地址本插件中

```
public void createPartControl(Composite parent) {
    //.....
    updateColumnNumbers();
    Activator.getDefault().getPluginPreferences().
        addPropertyChangeListener(
            propertyChangeListener);
}
```

```
private void updateColumnNumbers() {
    Preferences prefs = Activator
        .getDefault().getPluginPreferences();

    String columnNum = prefs.getString(
        PreferenceConstants.P_VIEW_COMBO);
    if(columnNum.equals("0"))
    {
        nameColumn.setWidth(200);
        categoryColumn.setWidth(0);
    }
    else if(columnNum.equals("1"))
    {
        nameColumn.setWidth(100);
        categoryColumn.setWidth(100);
    }
}
```

改变各列的宽度



在关闭视图时，删除添加的属性监听器

```
public void dispose(){
    //.....
    Activator.getDefault().getPluginPreferences().
        removePropertyChangeListener(propertyChangeListener);
    super.dispose();
}
```



图16-10 创建视图首选项的过程

图16-10的步骤中加入了updateColumnNumbers()方法，这是为了当程序启动时能够根据首选项页面中保存的值显示视图内容。

※ 注意 ※

代表插件的首选项值作为一个文件，被存储在下方的目录和文件中。

```
<workspace>\.metadata\.plugins\org.eclipse.core.runtime\.settings\com.plugindev.
addressbook.prefs
```

该文件被保存为工作空间的一部分，这样文件所定义的值仅对当前工作空间内部有效。当把首选项值写入磁盘时，对于那些当前值是默认值的键，实际上磁盘中并没有它们的记录。完成图16-10的操作后，保存的文件的内容如下所示。

```
#Mon Jun 25 10:38:44 CST 2007
eclipse.preferences.version=1
comboPreference=0
```

16.4.3 创建“编辑器”子首选项页面

本节将为编辑器创建子首选项页面，通过该页面定制编辑器的默认显示的页和默认初始布局。由于首选项的作用范围是整个工作空间，因而指定了默认显示页后，所有工作空间中打开的编辑器默认都将使用同一设置。

同“视图”首选项一样，首先需要在plugin.xml文件中声明该页面，并且创建继承了Field EditorPreferencePage，实现了IWorkbenchPreferencePage接口的EditorPrefPage类。在其中的createPartControl()方法中，创建三个布尔字段编辑器和一个选项编辑器，分别支持对编辑器页面的选择和“编辑”页面布局的选择。如图16-11所示。



图16-11 创建“编辑器”首选项页面



然后, 设置每个字段的标识和它们的默认显示值, 如下所示。

```
//PreferenceConstants
public static final String P_EDITOR_EDIT_BOOL = "editPageBooleanPreference";
public static final String P_EDITOR_PWSP_BOOL = "pwsPageBooleanPreference";
public static final String P_EDITOR_SOURCE_BOOL = "sourcePageBooleanPreference";
public static final String P_EDITOR_LAYOUT = "editPageLayoutPreference";

//PreferenceInitializer#initializeDefaultPreferences()
store.setDefault(PreferenceConstants.P_EDITOR_EDIT_BOOL, true);
store.setDefault(PreferenceConstants.P_EDITOR_PWSP_BOOL, true);
store.setDefault(PreferenceConstants.P_EDITOR_SOURCE_BOOL, true);
store.setDefault(PreferenceConstants.P_EDITOR_LAYOUT, "horizontal");
```

在编辑器的三个页面中, 只有“编辑”页面具有接收用户输入的功能, 因此此页面不能被去掉。如果希望当用户试图去掉此页面对勾时, 给出错误提示, 并且禁用“应用”和“完成”按钮, 需要通过覆写FieldEditorPreferencePage中的checkState()方法来实现。FieldEditorPropertyPage监听FieldEditor.IS_VALID属性更改事件, 并根据需要调用checkState()和setValid()方法。在EditorPrefPage类中加入如下的代码。

```
public void checkState(){
    super.checkState();
    if(!isValid())
        return;
    if(!boolEditFieldEditor.getBooleanValue()){
        setErrorMessage("\\"编辑\\"页面必须存在!");
        setValid(false);
    }
    else{
        setErrorMessage(null);
        setValid(true);
    }
}
```

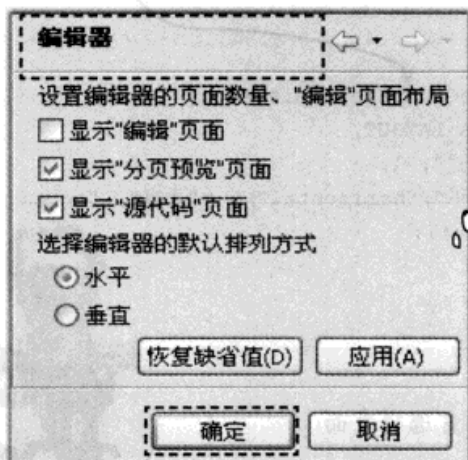
检查字段编辑器的状态是否有效

当“编辑”页面首选项的值为0时显示出错误信息

设置该首选项页面当前的状态为无效, 以此来禁用“应用”和“完成”按钮



看起来代码没有任何问题。加入上面的代码后, 重新运行地址本插件, 打开首选项页面, 将“编辑”页面对勾取消, 会发生什么呢?



无错误信息, 该首选项仍为有效。预料中的结果并未出现, 为什么?

•当取消“编辑”页面前的对勾时，不会显示出错信息。为何加入的代码无效？

名称过滤



一切都没有发生。原来，布尔字段编辑器永远不会处于无效状态，因而也就永远不会触发FieldEditor.IS_VALID属性更改事件。那么怎么办呢？还有一个办法。布尔字段编辑器当它的check框发生改变时，FieldEditor.VALUE属性更改事件会被触发。可以以此来捕获布尔字段编辑器值的更改。

为此，必须覆写FieldEditorPreferencePage中的propertyChange()方法，使在收到FieldEditor.VALUE属性更改事件时，调用checkState()方法，如图16-12所示。

```
public void propertyChange(
    PropertyChangeEvent event){
    super.propertyChange(event);
    if (event.getProperty().
        equals(FieldEditor.VALUE) ){
        if(event.getSource() ==
            boolEditFieldEditor)
            checkState();
    }
}
```

响应值更改事件

1.显示首选项页面错误信息
2.设置当前页面为无效状态

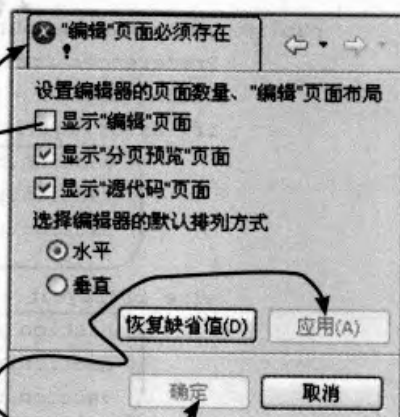


图16-12 检查“编辑”布尔字段编辑器的状态

接下来，修改地址本编辑器和“编辑”页的相关类，使其响应首选项的更改，如图16-13所示。

1. 响应“排列方式”首选项更改



在ScrolledPropertiesBlock类中声明属性监听器

```
private final IPropertyChangeListener propertyChangeListener =
    new IPropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent event) {
            if (event.getProperty().equals(
                PreferenceConstants.P_EDITOR_LAYOUT)){
                updateLayout();
            }
        }
    };
```

监听属性更改

改变布局

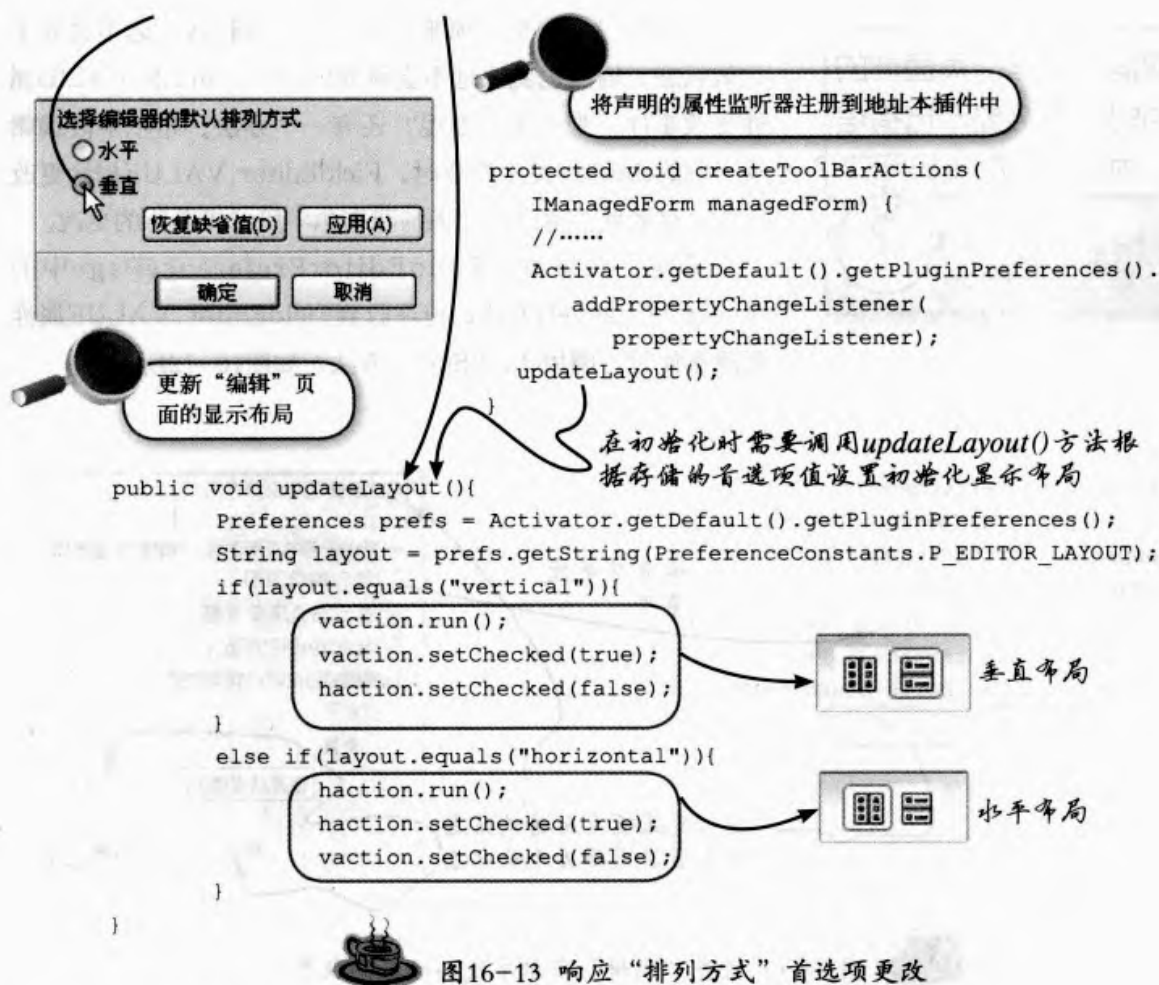


图16-13 响应“排列方式”首选项更改

2. 响应“显示页面”首选项更改

“显示页面”首选项更改的响应有些复杂，因为需要保证在增减页面的过程中，使每个编辑器页面的位置不变，如图16-14所示。



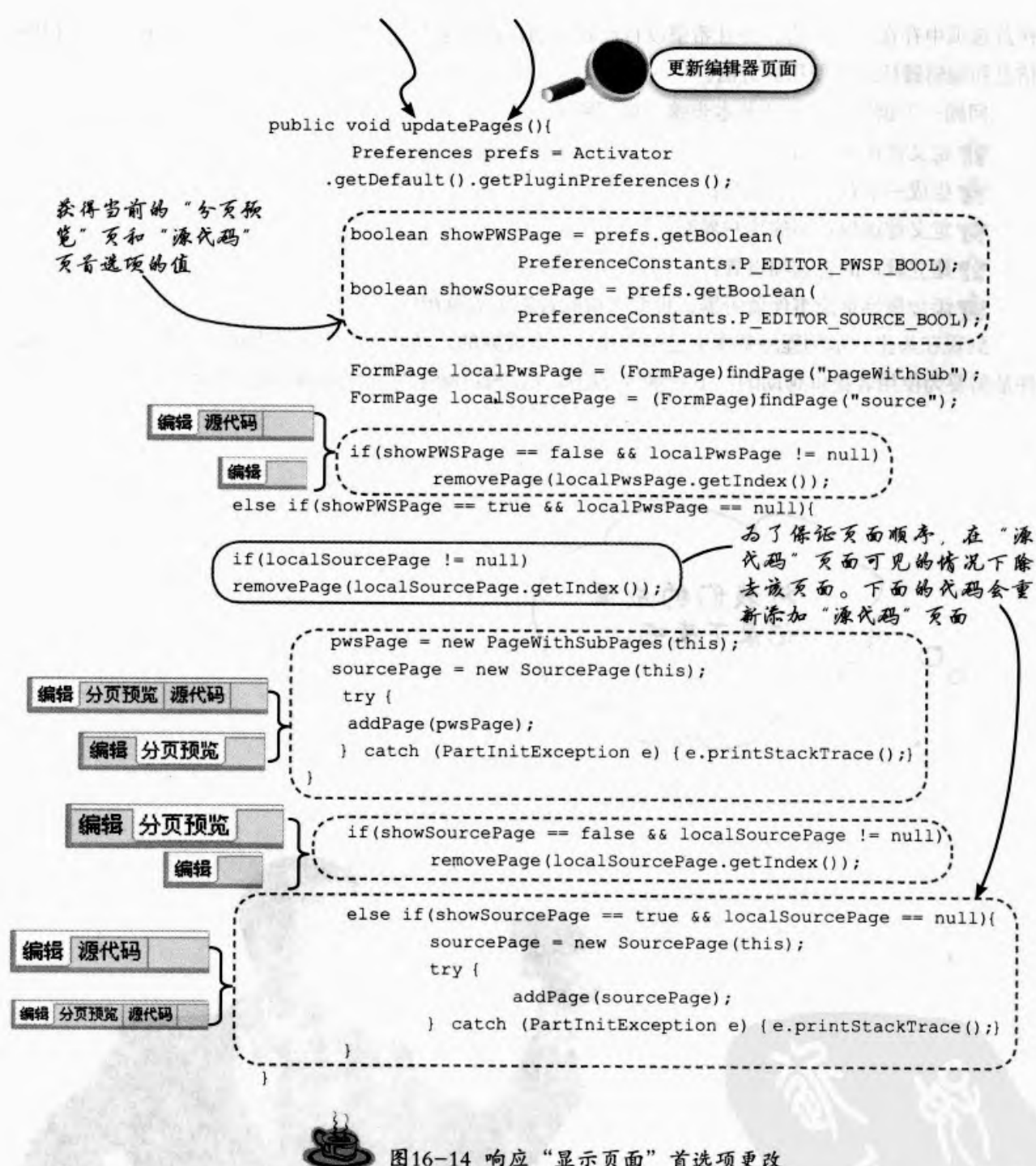


图16-14 响应“显示页面”首选项更改

16.5 本章小结

首选项页面允许用户定义并且保存他们喜欢的一些界面风格，这样，开发出的插件会更加人性化。Eclipse的首选项机制已经提供了存储功能，使开发者构造首选项时关注的方面更少。不过，如果

在首选项中存在很多内容，并且希望以自己定义的格式来保存首选项时，可以借鉴在保存地址本视图信息和编辑器信息中使用的方法。

回顾一下创建首选项的基本步骤，如下所示。

- ❶ 定义首选项页面扩展；
- ❷ 生成一个首选项页面类；
- ❸ 定义首选项页面的用户界面；
- ❹ 建立默认的首选项设置；
- ❺ 添加属性更改事件监听器，根据当前的首选项值做相应的改动。

到现在为止，示例程序基本上已经创建了一个成熟的，功能完善的地址本插件。但一个良好的插件是需要为用户提供帮助的。下一章将介绍如何在Eclipse中为地址本插件提供帮助内容。

将我们的成果
记录下来吧！



拉开崭新的学习帷幕

第17章 帮助内容 (Help Contents)

上一章介绍了如何为插件定义首选项内容，至此创建的地址本插件的主要功能都已经具备了，但是，如何让用户了解开发出来的插件的功能呢？使用过Eclipse的读者一定对Eclipse的帮助文档印象非常深刻，在许多人学习Eclipse的过程中，它起到了非常重要的作用。本章将介绍如何为Eclipse插件添加各种帮助内容，为用户使用插件提供有意义的帮助。

本章内容包括：

- ★ Eclipse用户辅助项目和帮助介绍。
- ★ 如何使用Eclipse帮助。
- ★ 使用Eclipse的帮助系统为应用程序提供集成化的文档。
- ★ 为地址本插件实现上下文相关帮助。
- ★ 如何在工作台的其他部分加入帮助按钮。



进入第17章

17.1 Eclipse帮助介绍

Eclipse帮助附属Eclipse的用户辅助项目，该项目主要包括三个内容，Eclipse欢迎框架，Eclipse备忘单和Eclipse帮助。本章将介绍Eclipse的帮助系统，下一章将为读者介绍Eclipse备忘单。对于Eclipse欢迎框架，本书将在介绍RCP程序时介绍。

Eclipse的用户辅助项目为使用Eclipse及其上开发的产品的用户提供辅助支持，用来帮助用户快速熟悉基于Eclipse产品的使用环境。无论对于哪一种软件来说，缩短用户学习曲线的能力对于该软件的推广及其能否取得成功都是至关重要的。EclipseUserAssistance项目的Web站点对Eclipse用户辅助项目的目标是“在Eclipse使用周期的所有阶段为该应用程序的用户提供帮助”。

在使用应用程序的许多时候，即使在业务或技术方面精通的人也不得不放低姿态，使用联机帮助文档来提供帮助。Eclipse的帮助是最方便最全面的Eclipse学习手册，特别是安装完中文语言包后的帮助文档，翻译得相当规范，用户可以通过它很快地学会如何使用Eclipse。

Eclipse把Help系统放在了与其他部分同等的地位，将其作为Eclipse平台的要素之一（参见第9章“Eclipse插件体系结构”）。一般的软件开发人员经常将开发帮助系统认为是无关紧要的工作，他们往往更加关注于具体功能的实现，而忽略用户使用的要求。他们认为这样的工作烦琐而又乏味，并不能创造价值。然而，事实恰恰不是这样，如果一个软件没有良好的帮助支持，它做得再好，由于对用户接受能力的要求，它的使用者范围也因此而受到限制。Eclipse平台将帮助系统集成到平台中，并且使提供联机帮助的操作简单易行，使插件开发者在将注意力集中于具体功能的同时，同样能够开发出优秀的帮助系统。就只是这样简单，只要除了帮助内容（这在任何平台下都需要由自己完成）外，所有需要提供的就只是一些简单的用于帮助的XML文件。

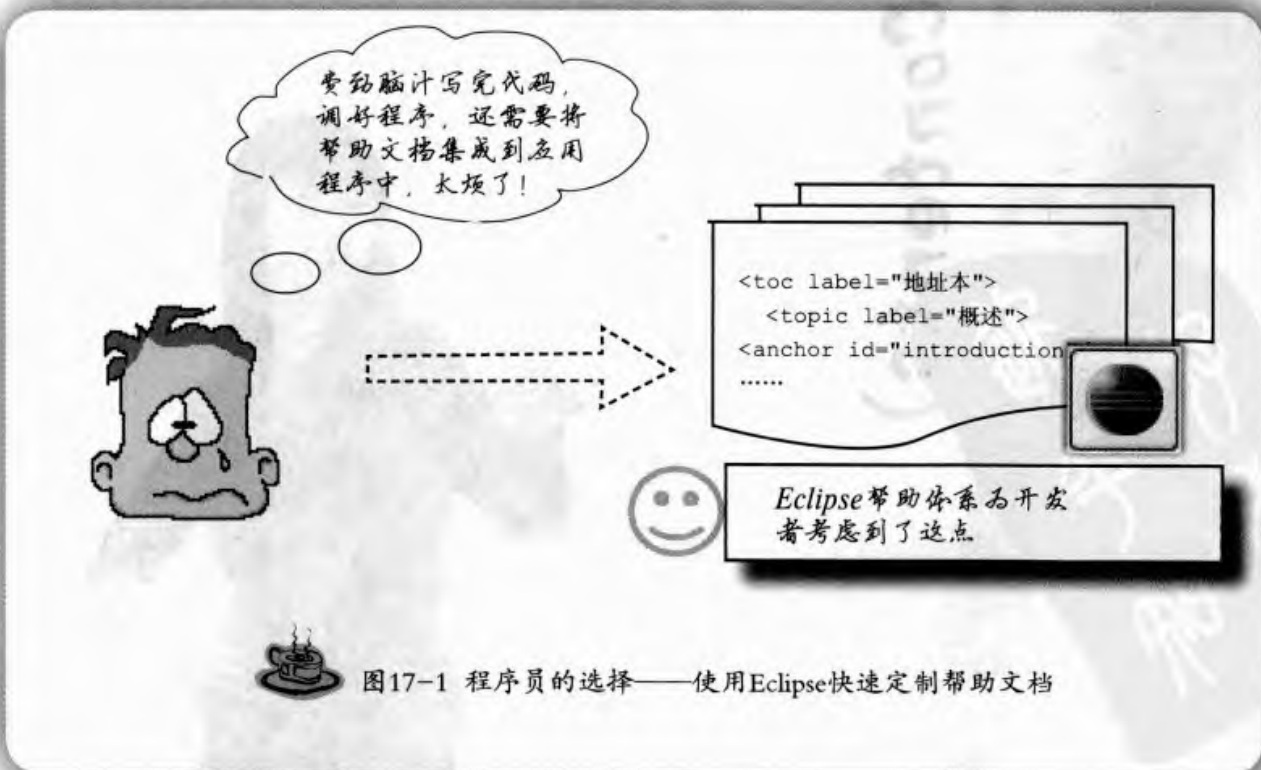


图17-1 程序员的选择——使用Eclipse快速定制帮助文档

17.2 使用Eclipse帮助

前面已经提到，Eclipse提供了功能强大的帮助系统。这可以通过选择“帮助”→“帮助内容”打开一个新的Eclipse窗口来访问整个Eclipse产品文档，也可以选择“帮助”→“动态帮助”或按F1键打开“帮助”视图。如图17-2所示。

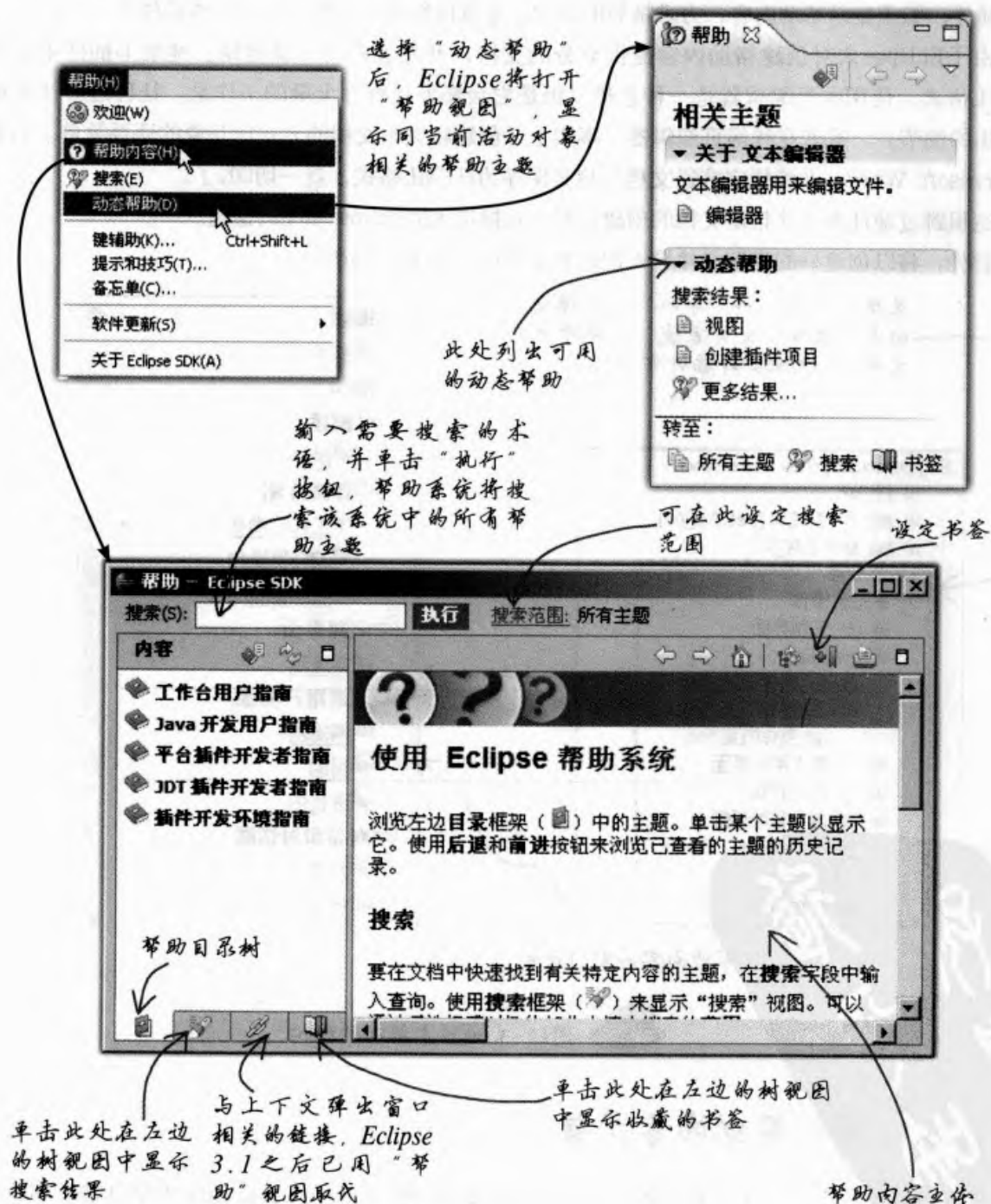


图17-2 使用Eclipse帮助

17.3 实现集成的帮助文档

Eclipse提供了一种将自己的帮助文档插入帮助系统的基本模式，这种模式使它不关注于帮助文件的创建（也应该如此，谁希望一个开发平台专门用来创建帮助文档呢！）。开发者可以为帮助系统提供HTML格式或PDF格式的帮助内容。插件开发者通常将帮助内容存放在单独的插件中，但也可存放到主插件。鉴于显示帮助内容，力求精简的目的，本章仍然将帮助集成到地址本插件中。

由于Eclipse未对创建帮助内容提供充分的支持，开发者需要另谋他路。地址本插件中选择了HTML格式。使用网页编辑器是一种选择（但是它会额外耗费你大量的工作量，让你过分注重编写HTML的细节），或者在线网页编辑器。事实上，创建超文本文档的方式比想象的还要简单。只要拥有Microsoft Word，在编辑完普通文档后将其保存为HTML格式，就一切OK了。

这里跳过地址本插件帮助文档的创建过程，直接进入同Eclipse相关的话题。

首先，将以创建好的帮助文件导入地址本插件中，如图17-3所示。

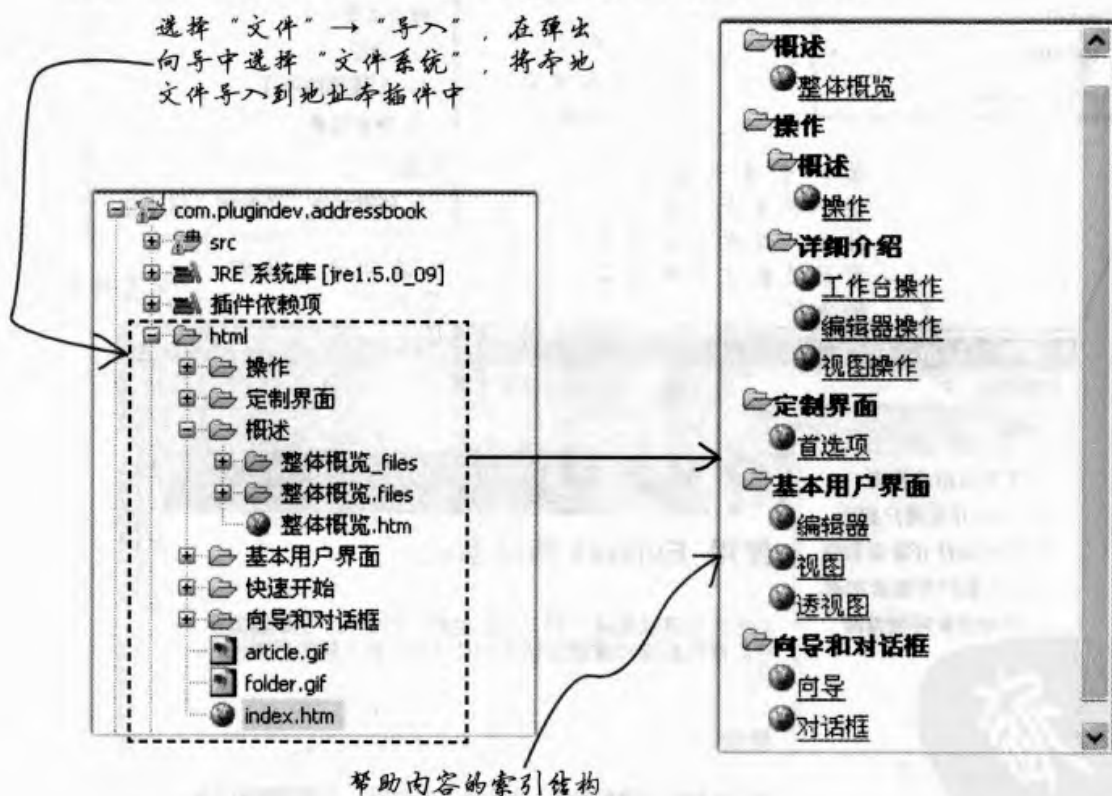


图17-3 地址本帮助的结构

17.3.1 添加帮助内容扩展

导入目录结构之后，接下来是将它们集成到Eclipse帮助系统中了。Eclipse本身自带了创建单独帮助插件或在插件中添加帮助内容的模板，可以使用它们来创建帮助示范，在此选择自己添加帮助扩展。打开清单文件，导航到“扩展”页面，单击“添加”按钮，在弹出的窗口中去掉“只显示必需插

件中的扩展点”前面的对勾，然后选择org.eclipse.help.toc扩展点，单击“完成”按钮。如图17-4所示。

去掉该选项前面的对勾，
否则，该扩展点不会在扩展点列表中显示

可以从此处看到toc的含义：
Table of Contents
单击此链接可以打开

如果需要为扩展点创建模板，选中该项，
系统将为插件生成帮助样例

选中该扩展点后，单击“完成”按钮，会弹出提示对话框

选择“是”，在创建了新的扩展的同时，地址本插件所依赖的插件中也将导入org.eclipse.help插件

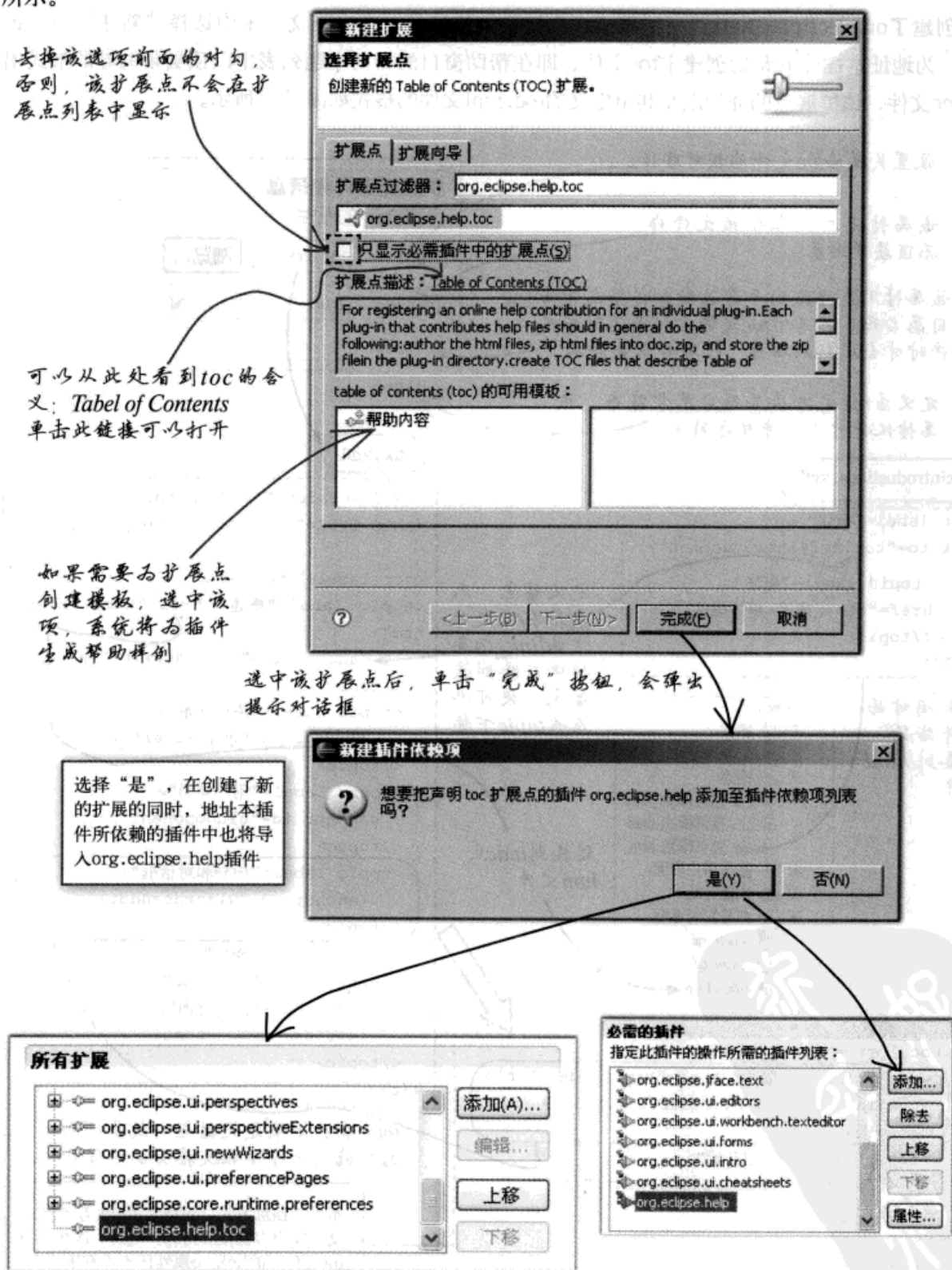


图17-4 为地址本添加org.eclipse.help.toc扩展

17.3.2 添加帮助内容

创建了org.eclipse.help.toc扩展后,右键选中该扩展,在上下文菜单中选择“新建”→“扩展”,为地址本插件的帮助创建主toc文件,即在帮助窗口的帮助主题列表中以顶层帮助卷的形式出现的toc文件,该扩展点的介绍以及其中定义的toc.xml文件的格式如图17-5所示。

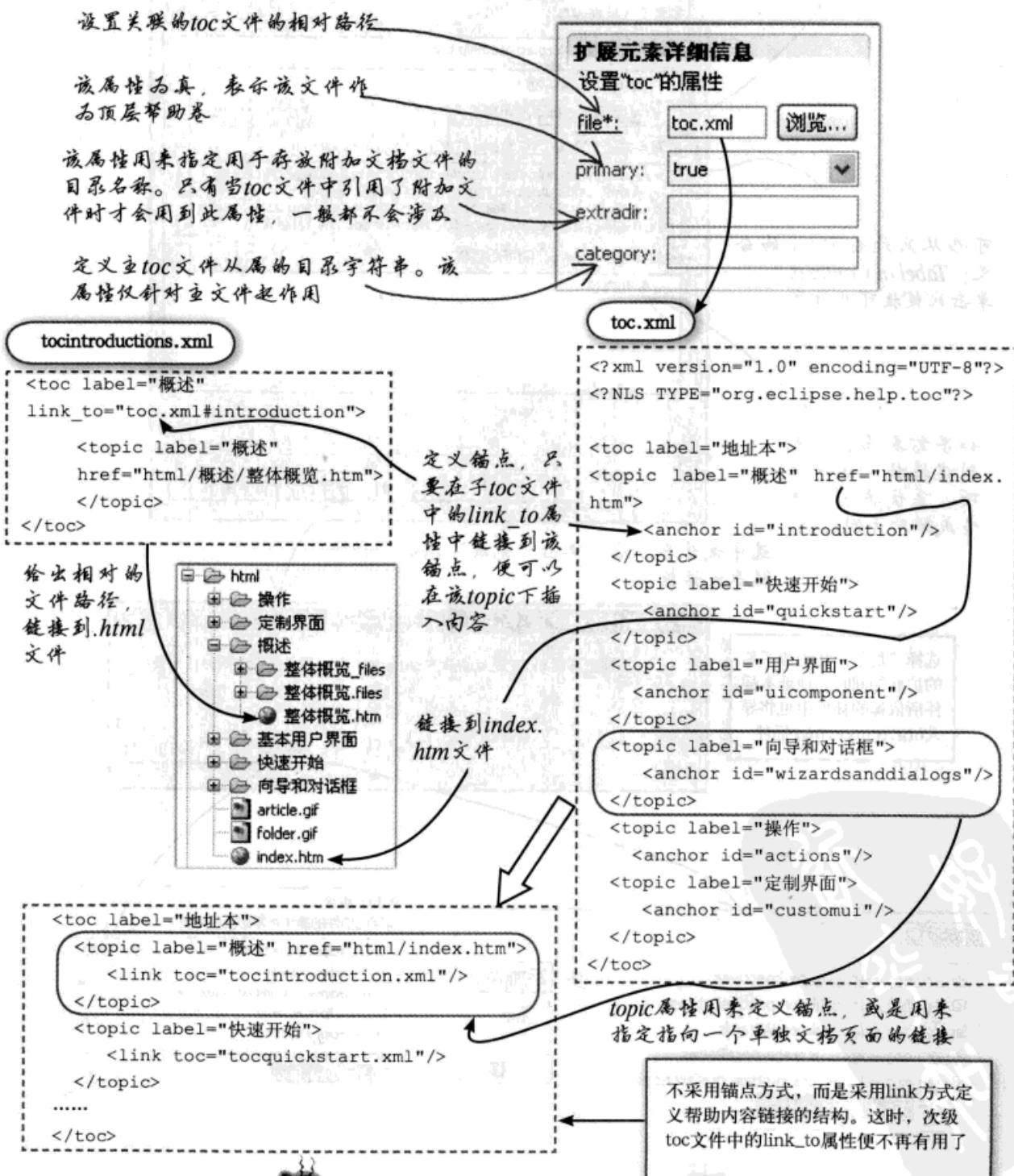


图17-5 toc扩展点及toc.xml文件格式介绍

在图17-5中容易看到，可以使用两种方式来组织帮助内容，一种是使用锚点的方式，这种方式同Eclipse插件体系结构的组织方式是一致的，上层toc文件提供的锚点实际上就是允许下层toc文件插入的位置，它不必关心到底有没有往该锚点中插入的文件，如果有的话也不需要知道有多少个下层toc文件。这提供了帮助文档组织上的极大灵活性，并且随着时间的增长，来自于插件内外的附加资料能够被有机地添加到帮助文档中。

但是如果要保护帮助文档，不希望它们被其他插件随意地加入内容该怎么办呢？这就需要使用链接的方式。对于链接的方式，上层toc文件需要提供下层toc文件的链接位置，也就是说，上层toc文件需要感知到下层toc文件的存在，没有上层toc文件的允许，下层toc文件是不可以插入到上层定义的帮助文档结构中的。Eclipse提供了这两种不同的方式，具体采用什么方式需要开发者来做出选择，因为，只有开发者本人才知道自己需要什么。如图17-6所示。

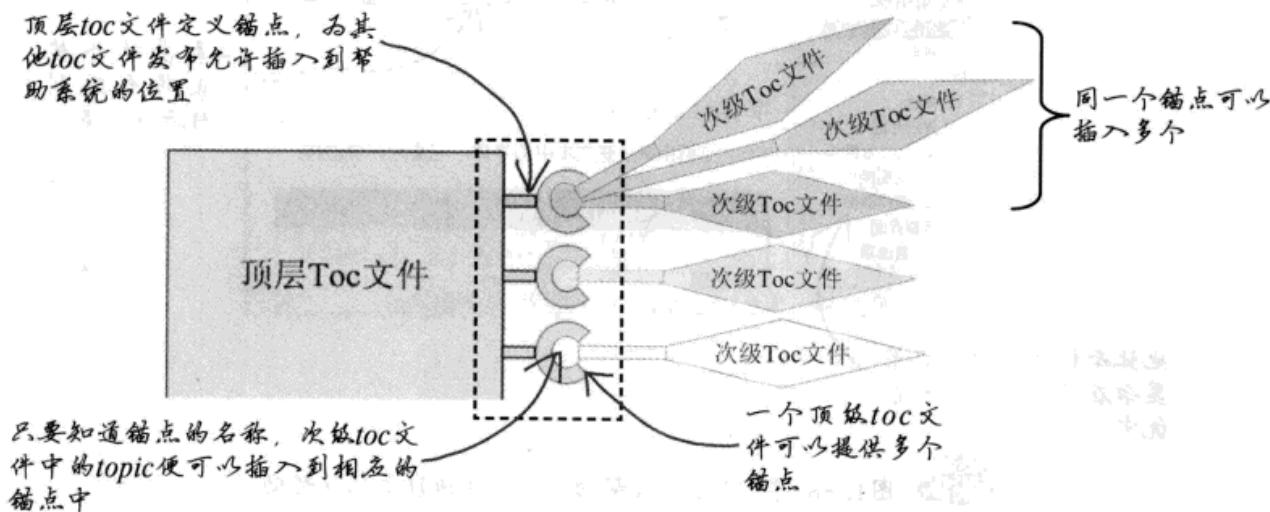


图17-6 采用锚点的方式构建Eclipse帮助体系

地址本插件中的toc文件只有两级结构，顶级toc文件中定义了6个锚点，本章的示例中为其中的5个添加了内容，对于“quickstart”锚点，将在下一章介绍备忘单时再充实其内容。在tocaction.xml文件中，示例程序采用了嵌套的topic来描述目录的不同级别，如图17-7：

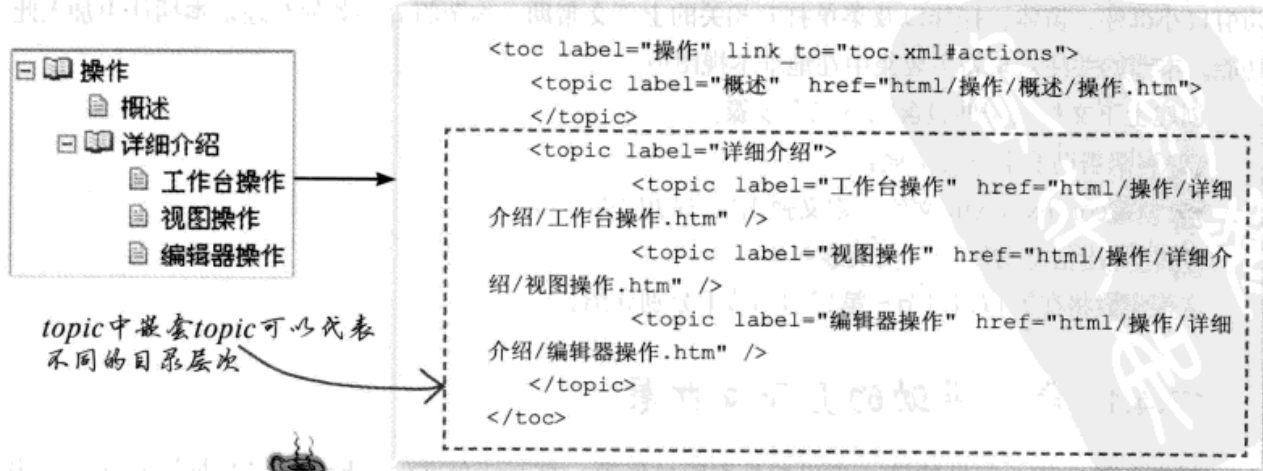


图17-7 次级toc文件对目录结构的一种表示方法

其他的toc文件同图17-5中介绍的tocintroductions.xml文件基本相同,在此不做一一介绍了。

完成所有的toc文件同HTML文件的绑定工作后,运行地址本插件,选择“帮助”→“帮助内容”,地址本插件的帮助将会出现在Eclipse的帮助系统中,如图17-8所示。

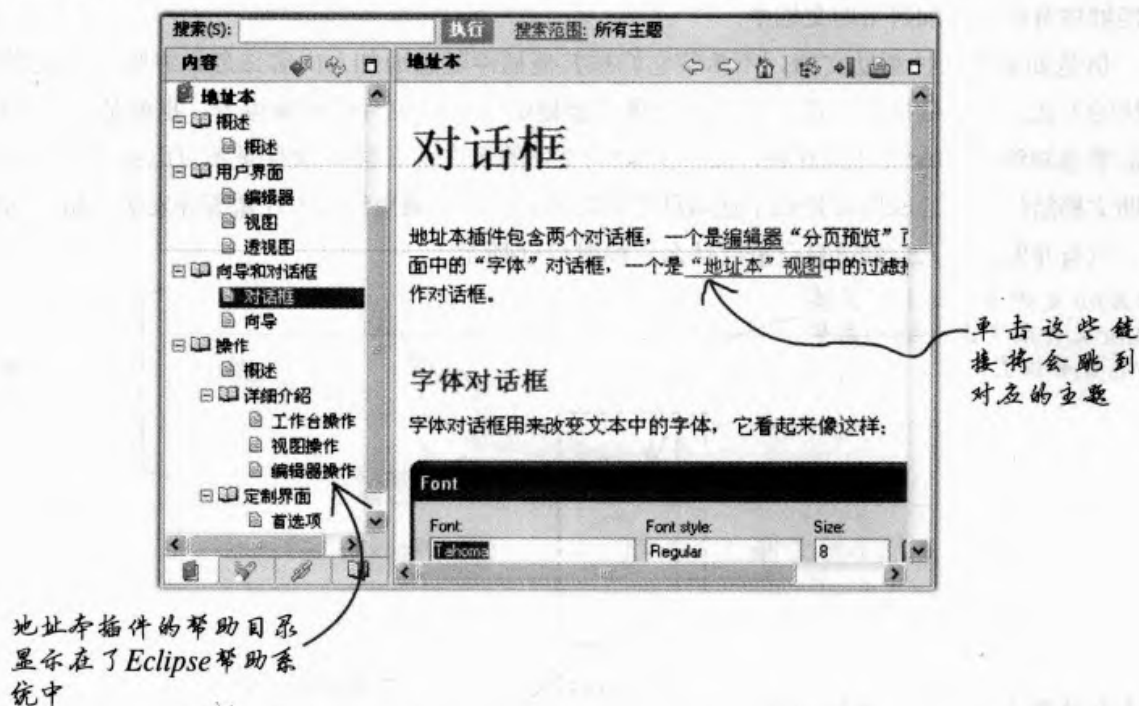


图17-8 集成到Eclipse帮助系统中的地址本插件帮助

17.4 上下文相关帮助

第17.1节提到了与当前活动对象相关的上下文相关帮助。Eclipse的帮助系统支持使用F1功能键,为窗口小部件、窗体、操作以及菜单打开相关的上下文帮助。本节将介绍如何在地址本插件中加入此功能。本节的关注对象将主要集中在地址本视图中。

创建上下文相关帮助包含以下三个步骤。

1. 声明帮助的上下文扩展;
2. 创建contexts.xml文件,定义弹出信息的内容;
3. 把弹出信息与UI上下文关联起来。

这些步骤将在第17.4.1节~第17.4.3节中分别介绍。

17.4.1 声明帮助的上下文扩展

创建上下文相关帮助的第一步仍然是声明扩展项。打开清单文件,导航到“扩展”页面,单击

“添加”按钮，在弹出的窗口中选择org.eclipse.help.contexts后，单击“完成”按钮。然后为该扩展添加contexts元素。

contexts元素具有两个属性，file属性提供描述contexts的XML文件相对于插件的路径，plugin属性指示插件的ID。帮助系统通过上下文的ID来识别不同的上下文，这些上下文标识的格式一般是<plugin-id>.<local_context_id>，其中的<plugin-id>便是此处plugin属性中定义的插件标识字符串。

设置file属性的文件名为context.xml，plugin属性的名称为com.plugindev.addressbook，如图17-9所示。

17.4.2 定义弹出信息内容

在此仍然将注意力集中到视图中，为视图中的相关元素定义弹出信息内容。首先在地址本插件根目录下创建插件清单中定义的文件contexts.xml，在其中加入上下文帮助的相关内容，其格式如图17-9所示。

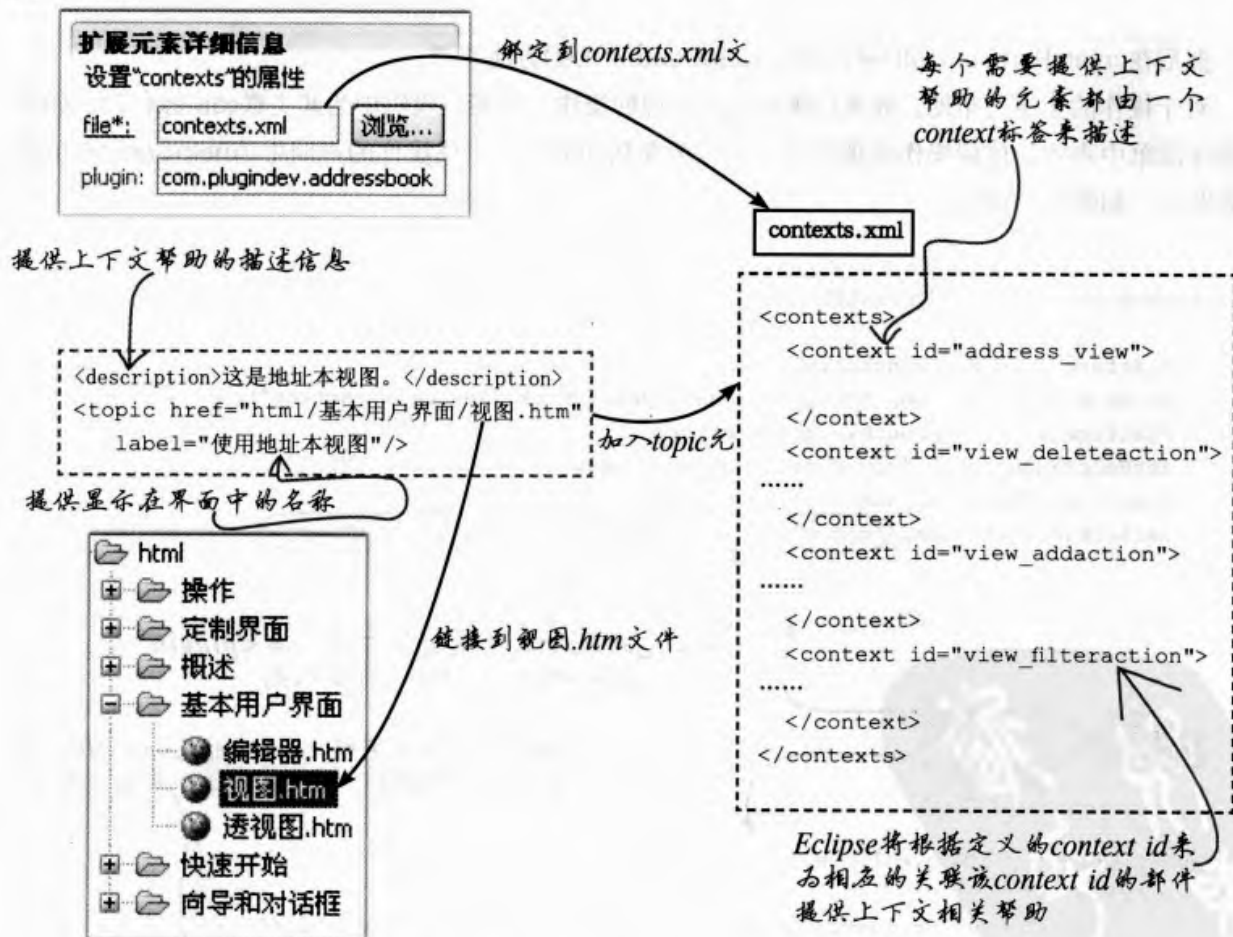


图17-9 定义contexts.xml文件中的内容

示例程序声明了4个context元素，分别为地址本视图、视图中删除操作、添加操作和过滤操作添加上下文相关帮助。其他context元素中的内容同address_view中的类似，这里不再赘述。

17.4.3 关联弹出信息同UI上下文

上一节创建了contexts.xml文件, 并提供了代表UI元素的context_id, 这些context_id如何同具体的UI元素相关联的呢? 本节将讨论这个问题。

Eclipse的帮助系统提供了IWorkbenchHelpSystem.setHelp(Control, string)来为窗口部件提供context id。获得工作台帮助系统的实例可以使用以下方法。

```
PlatformUI.getWorkbench().getHelpSystem();
```

可以通过在AddressView.java中加入setHelpContextIDs()方法来达到目的, 如下面的代码所示。

```
private void setHelpContextIDs(){
    PlatformUI.getWorkbench().getHelpSystem().
        setHelp(viewer.getControl(),
            "com.plugindev.addressbook.address_view");
}
```

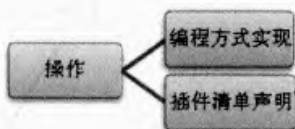
需要给出context ID的全称

然后在createPartControl()中调用setHelpContextIDs()方法即可。

对于操作的上下文帮助, 如果以编程方式实现的操作, 需要在代码中为其关联context id, 如果在插件清单中声明, 使用操作代理实现, 则只要在插件清单关于该操作的声明中为helpContextId加入值即可, 如图17-10所示。

```
private void setHelpContextIDs(){
    .....
    PlatformUI.getWorkbench().getHelpSystem().
        setHelp(deleteAction, "com.plugindev.addressbook.view_deleteaction");
    PlatformUI.getWorkbench().getHelpSystem().
        setHelp(addAction, "com.plugindev.addressbook.view_addaction");
    PlatformUI.getWorkbench().getHelpSystem().
        setHelp(filterAction, "com.plugindev.addressbook.view_filteraction");
}
```

此处给出的是id的全称, 为<plugin-id>.<local_context_id>的形式



在操作的扩展点声明中加入helpContextId, 为“打开地址本视图”操作添加上下文相关帮助

```
<extension point="org.eclipse.ui.actionSets">
  <action
    class="com.plugindev.addressbook.actions.OpenAddressViewAction"
    definitionId="com.plugindev.addressbook.commands.openAddressView"
    helpContextId="address_view"
    icon="icons/sample.gif"
    id="com.plugindev.addressbook.actions.OpenAddressViewAction"
    label="打开视图"
    menubarPath="AddressBookMenu/content"
  />
</extension point>
```

```

        toolbarPath="Normal/addition"
        tooltip="在当前透视图中打开地址本视图"/>
    </actionSet>>
</extension>

```



图17-10 将操作关联到context_id的方法

以上是关联视图同上下文ID的方法，对于其他Workbench部件，关联的方法相类似，都是在createPartControl()中（或相应的其他方法，如对于对话框中的上下文帮助，在createDialogArea()方法中）加入PlatformUI.getWorkbench().getHelpSystem().setHelp()方法。

事实上，对于继承Action类的操作类，还有另外一个方法可以使用。Action类提供了setHelpListener()方法，该方法能够把一个帮助监听器同该操作关联起来。如对于图17-10中提到的addAction对象，可以使用以下方法来达到相同的目的。

```

addAction.setHelpListener(new HelpListener(){
    public void helpRequested(HelpEvent e){
        PlatformUI.getWorkbench().getHelpSystem().
            displayHelp("com.plugindev.addressbook.view_addaction");
    }
});

```

当按下F1键时，帮助监听器收到一个帮助事件，helpRequested()方法将会处理该事件

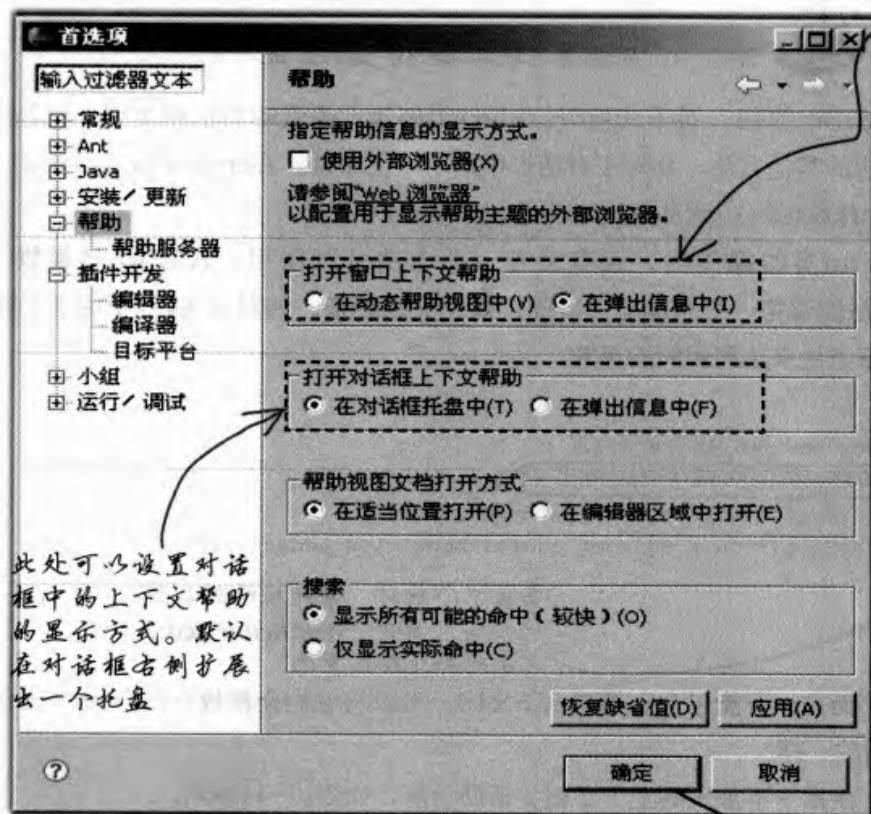
对于非常复杂的应用程序而言，可能包含大量的上下文ID，可以将它们全都放在一个.java文件中管理，就像对图像文件所做的一样。

现在，运行地址本插件，使用一下加入的上下文相关帮助功能，如图17-11所示。



图17-11 上下文相关帮助的效果图

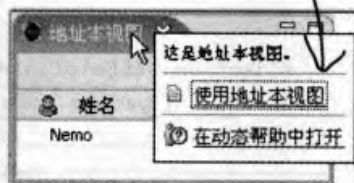
Eclipse在3.1版本之后，默认使用视图的方式显示上下文相关的帮助，但是在3.1版本之前，是使用弹出信息框来显示上下文相关帮助的。如果希望在3.1版本之后用弹出信息框来显示上下文相关的帮助，可以在“首选项”中进行如图17-12所示设置。



在此处将打开窗口上下文的帮助设为“在弹出信息中”，则上下文相关的帮助将会出现在弹出信息中

在上下文帮助信息中弹出上下文帮助

此处可以设置对话框中的上下文帮助的显示方式。默认在对话框右侧扩展出一个托盘



单击确定后，选择F1



图17-12 设置“首选项”

17.5 加入帮助按钮

除了选择“帮助”→“帮助内容”之外，Eclipse还允许使用其他方式打开帮助链接。只要使用`IWorkbenchHelpSystem.displayHelpResource(String)`，并且提供要打开的HTML文件的href即可。下面演示在地址本视图的工具栏中加入帮助按钮，并将其链接到帮助系统中的“视图”页面中的方法。

添加帮助按钮

```
private void addHelpButtonToToolBar(){
    Action helpAction = new Action(){
        public void run(){
            PlatformUI.getWorkbench().getHelpSystem().displayHelpResource(
                "/com.plugindev.addressbook/html/基本用户界面/视图.htm");
        }
    };
}
```

在单击“帮助”按钮后，将会打开帮助窗口，显示指定的帮助页面



17.6 本章小结

当使用Eclipse开发产品时，使用帮助文档能够使用户上手更为快速。同样，基于Eclipse开发出来的插件也需要为用户提供帮助方便用户的操作。帮助作为Eclipse的许多可用的集成点之一，允许开发者快速地开发出集成的帮助文档。Eclipse也为开发者提供了上下文相关帮助的支持，允许对当前关注的焦点提供帮助，这更加有利于快速锁定用户遇到的问题。

虽然Eclipse要求用户具有一致性的体验，例如规定使用F1键来显示上下文相关的帮助，但仍然给了开发者很大的自由。毕竟，开发者可以定义帮助的内容以及它的显示方式。

如果帮助文档比较多，可以将帮助打包成ZIP格式，并且将其命名为doc.zip，Eclipse系统会自动识别该ZIP包。

下一章将会关注另外一种用户辅助部件——备忘单，它以独特的方式为用户提供学习教程。

将我们的成果
记录下来吧！



第18章

备忘单 (CheatSheet)

拉开崭新的学习帷幕

上一章中介绍了如何为Eclipse插件提供帮助文档，这一章将介绍一种更为强大和神奇的用户辅助设施——Eclipse备忘单。Eclipse备忘单通过自动创建用户所需要的命令来手把手地教会用户如何使用插件。这是Eclipse为开发者提供的新奇而友好的帮助手段，因而，如果能够使用这种手段来为用户提供相应的教程，将会对推广插件非常有帮助。本章将通过为地址本插件提供备忘单介绍备忘单的创建和使用。

本章内容包括：

- ★使用Eclipse备忘单。
- ★为地址本插件创建备忘单。
- ★构建复合备忘单。
- ★关联备忘单到帮助文档。



进入第18章

相信很多读者可能会有这样的经历（说不定在阅读本书的时候便是这样），当学习一门新的技术或熟悉一个新的软件的时候，一边阅读同该技术或软件相关的文档，一边使用文档中提供的方法在自己的计算机上进行操作。相信如果有这样的经历的话，一定会苦不堪言（笔者个人便是如此）。在学习的过程中注意力不得不在不同的东西之间来回的切换——特别是非电子版的文档时。这样的时代应该结束了！Eclipse提供给了插件开发者，也使插件开发者能够提供给用户更为有效的学习方式，这就是备忘单！备忘单机制使Eclipse可以为用户提供某个视图，告诉用户下一步要执行什么，怎么执行，甚至自动为其执行任务。它是如此的神奇，开发方式又是如此的简单，相信读者马上便会体会到这一点。如图18-1所示。

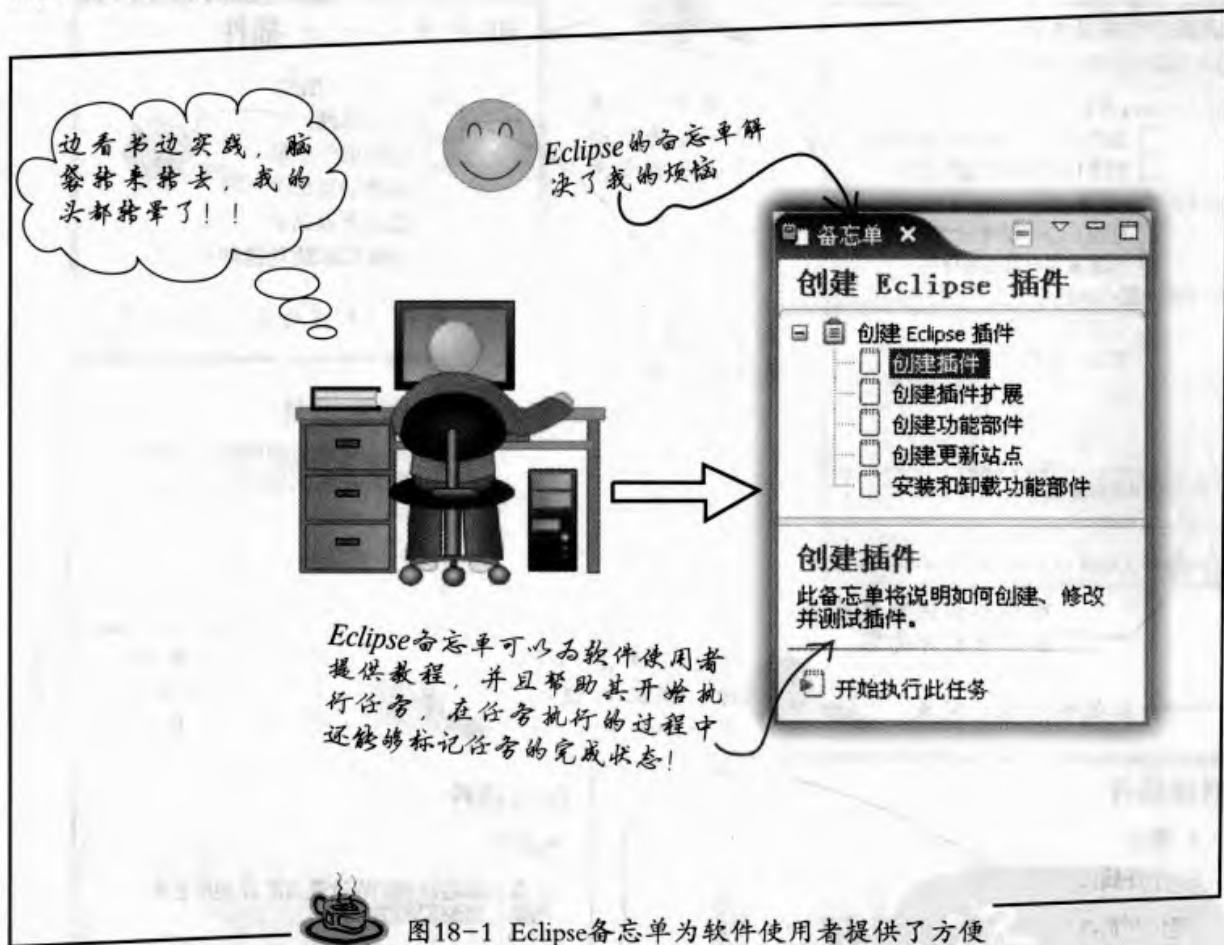


图18-1 Eclipse备忘单为软件使用者提供了方便

备忘单是 Eclipse 3.0 中新出现的技术，这种技术能够引导开发人员执行一系列复杂任务，从而完成某一总体目标。某些任务可以自动执行，如替用户启动所需的工具，其他任务需要由用户手工完成。备忘单任务可以与 Eclipse 帮助挂接，这样就不再需要搜索文档了。可以在 Eclipse 中“帮助”→“备忘单”下面找到一些参考的备忘单示例。图18-1所示的便是Eclipse 3.2.1自带的“创建Eclipse插件”复合备忘单示例。

本章将继续完善地址本插件的功能，为地址本插件开发一个新的备忘单。示例程序将从简单的备忘单开始，逐步构造出一个能够使用变量定义重复项，能够自动执行任务的复合备忘单。在本章最后，会将备忘单链接到“帮助内容”中，使它们作为Eclipse用户辅助设施共同为用户提供方便。

18.1 使用Eclipse备忘单

当插件开发者在进行一项新的任务时,迫切希望能够很快地上手,又比较反感庞杂的帮助系统扑面而来的巨量信息和生硬的文字时,可以首先查看一下是否Eclipse系统中提供的备忘单可以满足这种较为苛刻的要求。Eclipse 3.2中提供了Java开发、插件开发和小组CVS三个主题的备忘单,由这些备忘单入手,将会大大缩短入门的时间。如图18-2所示演示了使用“创建Eclipse插件”的过程。

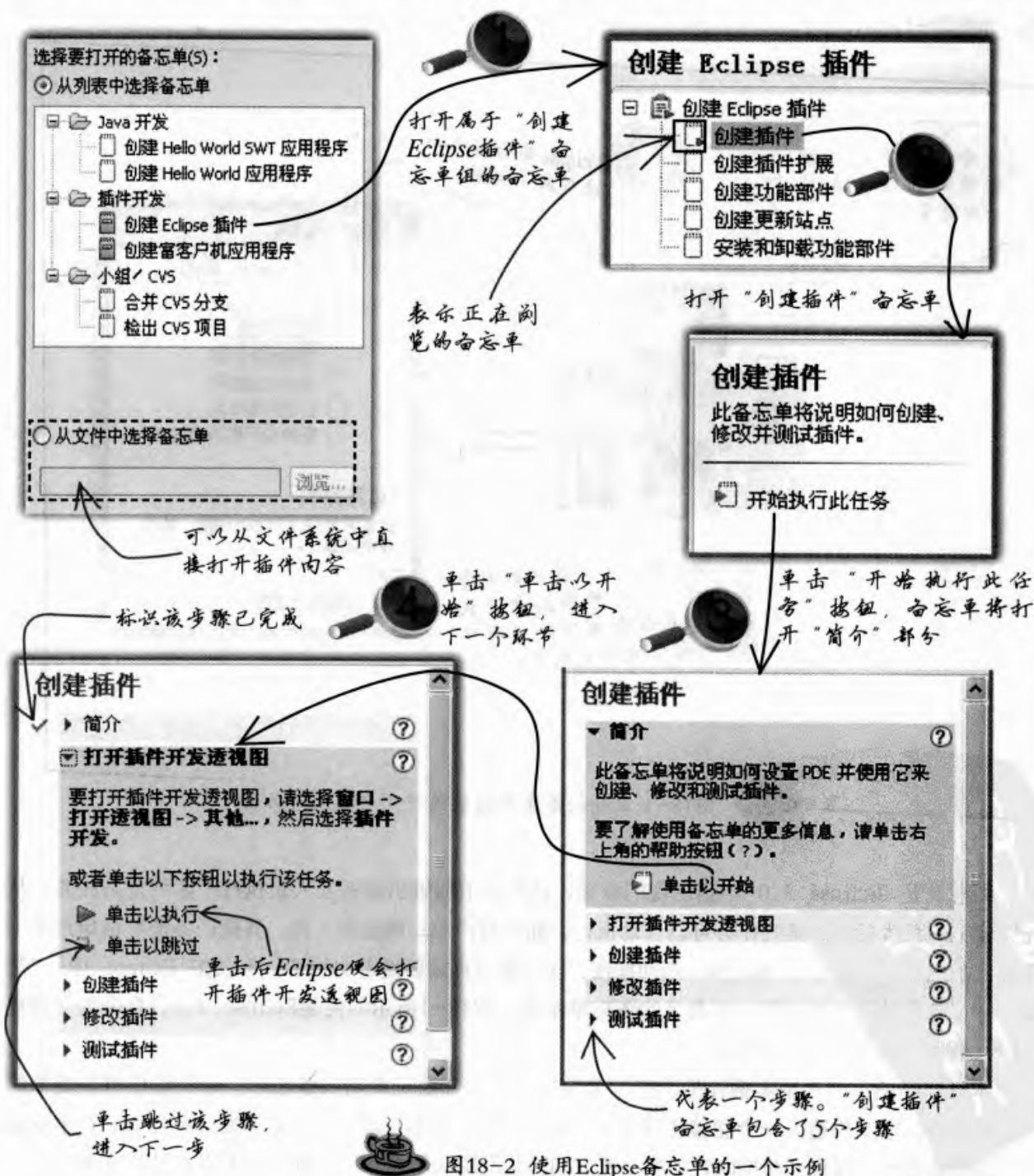


图18-2 使用Eclipse备忘单的一个示例

18.2 为地址本插件创建备忘单

创建地址本备忘单需要使用Eclipse提供的org.eclipse.ui.cheatsheets.cheatSheetContent扩展点，这个扩展点可以用来注册新的备忘单。为了创建扩展，进入清单编辑器底部的“扩展”选项卡并单击“添加”按钮。在弹出的“新建扩展”向导对话框中，取消底部“只显示必需插件中的扩展点”的复选框并选择 org.eclipse.ui.cheatsheets.cheatSheetContent 扩展点，然后单击“完成”。如果询问是否将 org.eclipse.ui.cheatsheets 插件添加到依赖项列表中，那么选择“是”。这样便将该扩展点加入了清单中。如图18-3所示。

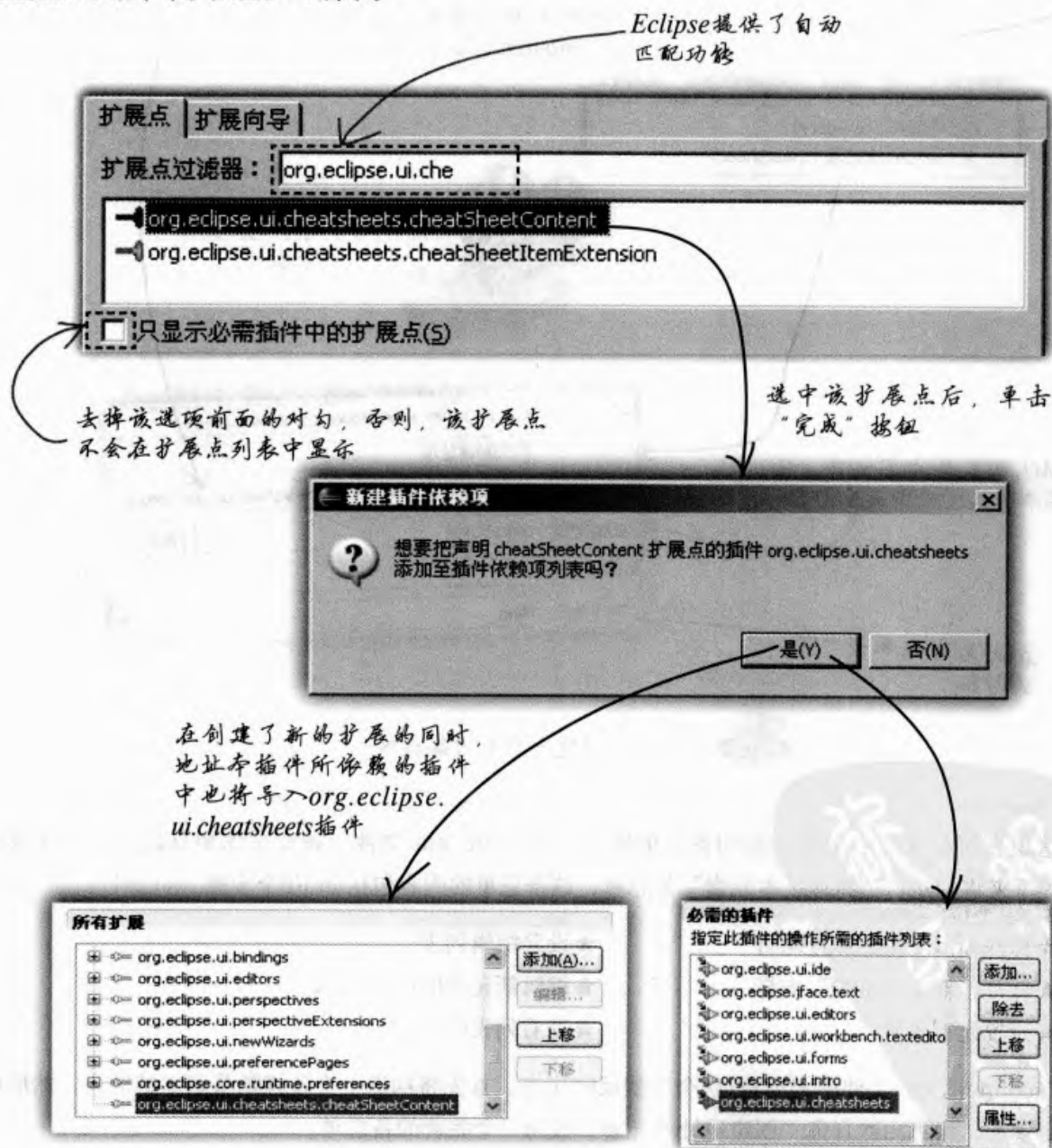
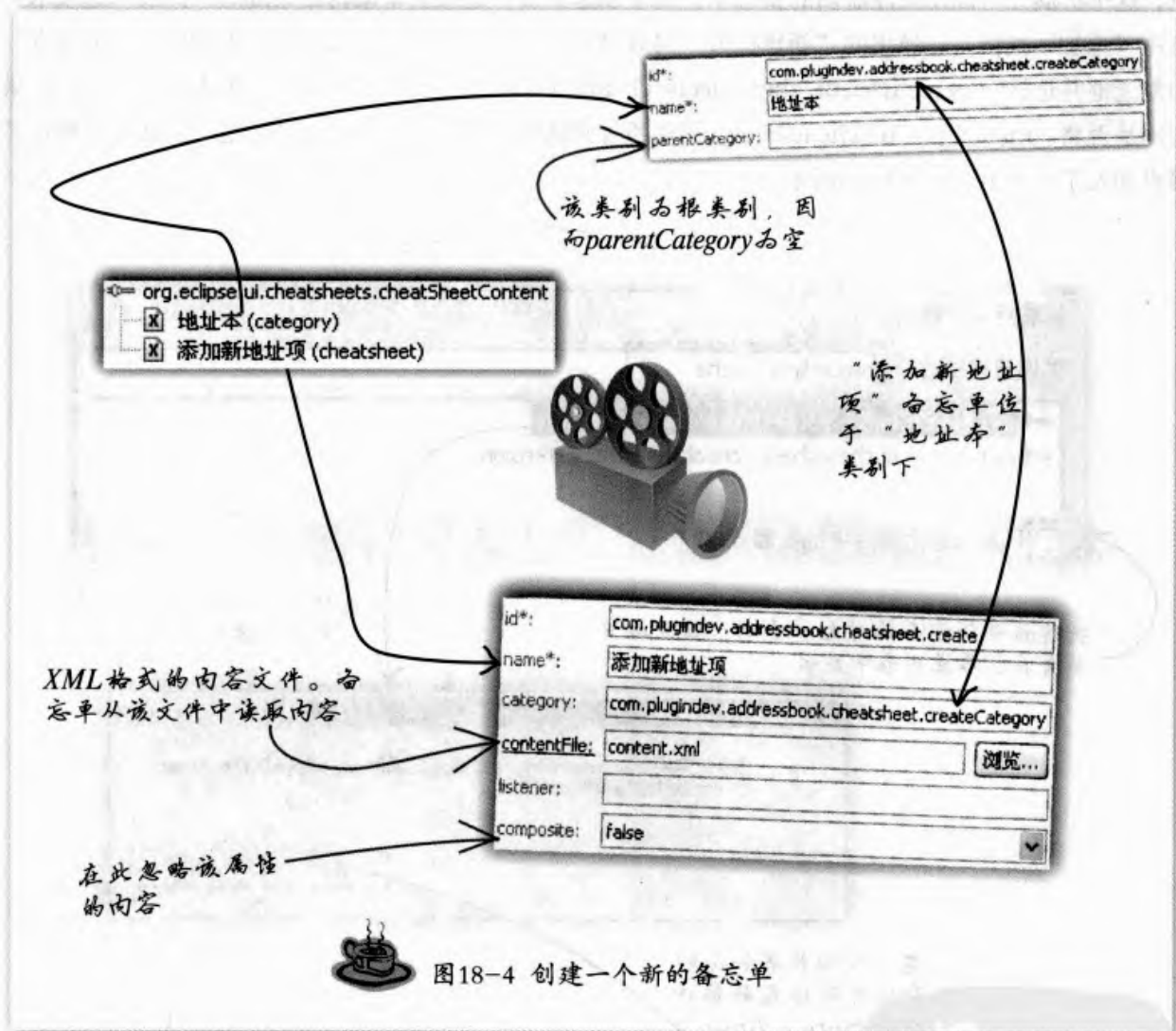


图18-3 使用备忘单内容扩展点

添加了该扩展后，选中备忘单内容扩展，单击右键，在上下文菜单中选择“新建”→“category”，为备忘单创建一个新的类别，在其中输入类别标识和名称。然后再右键单击该扩展，在上下文菜单中选择“新建”→“cheatsheet”，创建一个新的备忘单项，在其中填入如图18-4的内容。



这里在清单文件中为新创建的备忘单指定了content.xml文件，备忘单需要从此文件中读取内容。接下来将创建“创建地址本元素”备忘单，该备忘单的内容包括如下6个步骤 (steps)。

- ★打开AddressBook透视图
- ★设置编辑列表
- ★打开“新建地址项”向导
- ★编辑新元素的内容
- ★设置元素属性
- ★查看列表内容

在content.xml文件中需要将这6个步骤反映出来。首先将构造一个备忘单最基本的部分，然后由简单到复杂，增加功能直到“创建地址本元素”成为一个完善的备忘单。

选择“文件”→“新建”→“文件”，在地址本插件中创建一个名为content.xml的文件夹。打开该文件夹，在其中按照备忘单的格式写入XML代码，如图18-5所示。

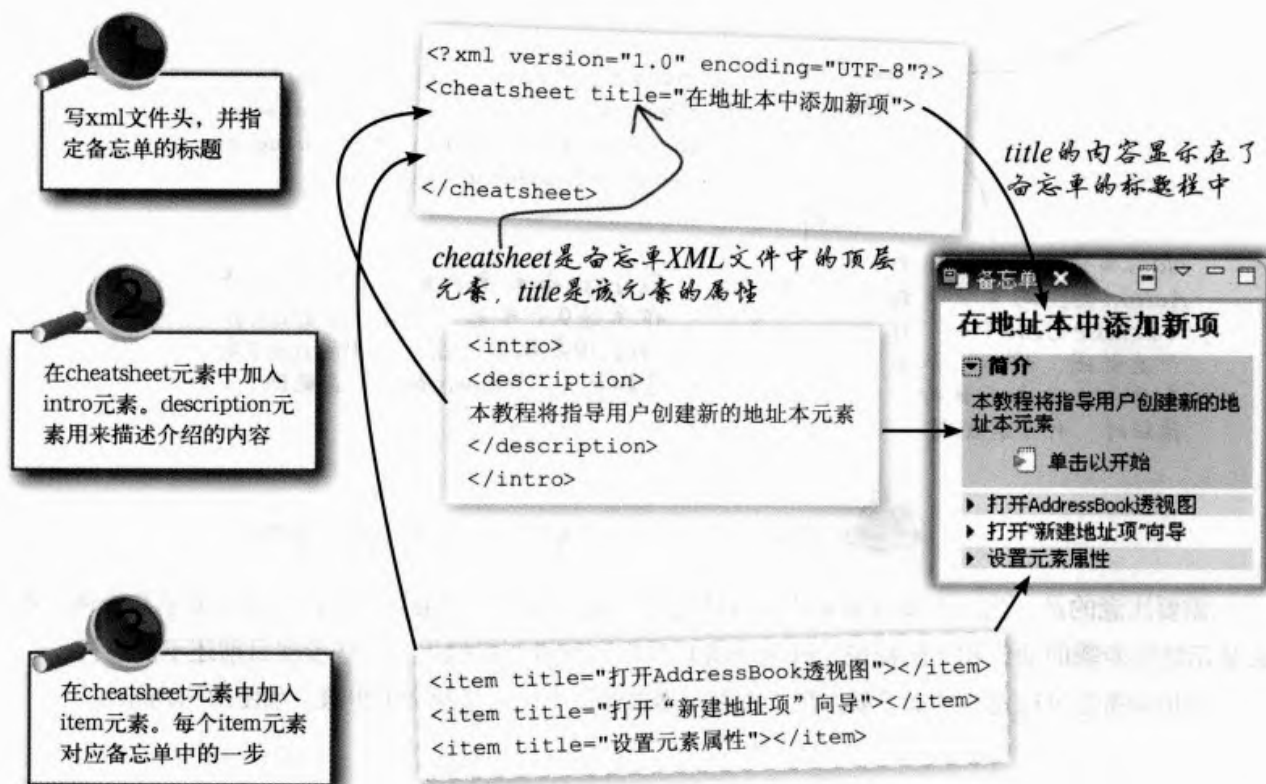
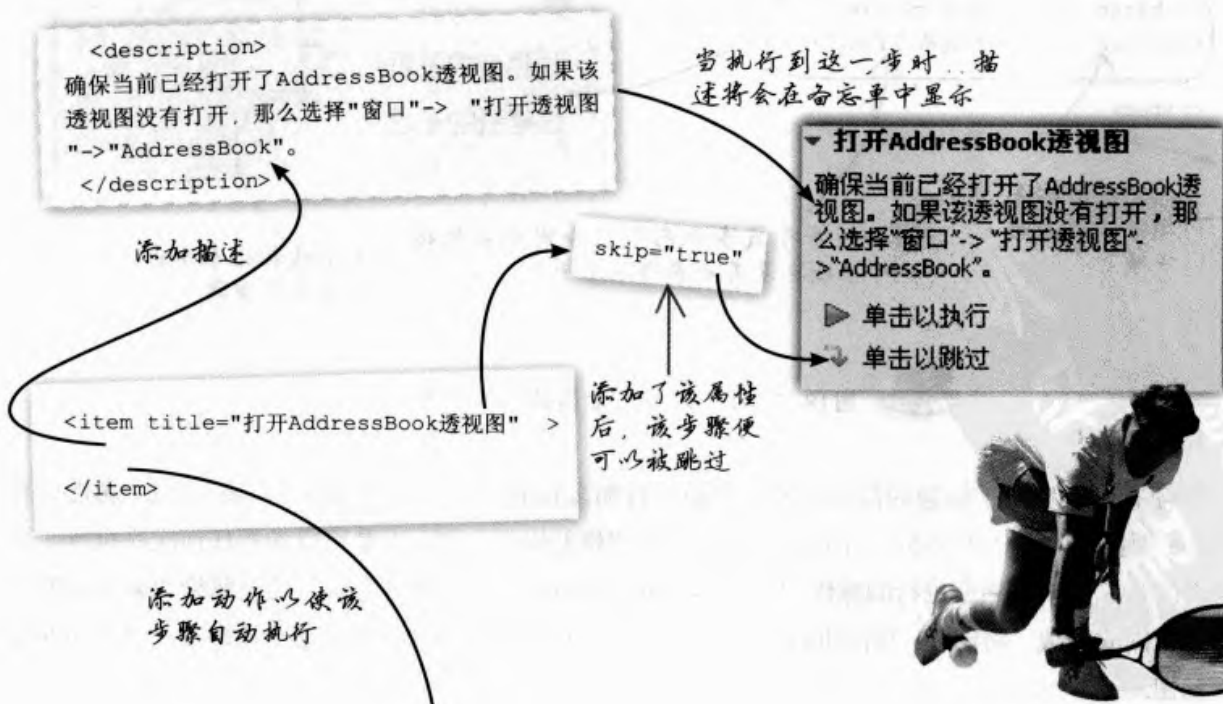


图18-5 创建备忘单的基本组成部分

完成这三部分后，需要充实备忘单的每一步。以“打开透视图”为例，为该步骤填充内容。图18-6将该步骤关联到Eclipse操作中，从而实现备忘单中该步骤的自动执行。



class属性对应到相应的Action类。这里关联到了Eclipse UI自带的“打开透视图”操作，从而在单击“单击以执行”按钮时，打开透视图

```
<action pluginId="org.eclipse.ui.cheatsheets"
class="org.eclipse.ui.internal.cheatsheets.
actions.OpenPerspective" param1="com.plugindev.
addressbook.AddressBookPerspective"/>
```

Eclipse备忘单机制可以允许为操作传递9个参数，分别以param1-param9来标识。在此处用param1来告知打开“AddressBook”透视图



图18-6 填充“打开AddressBook透视图”步骤

需要注意的是，当一个步骤未完成或未跳过时，它之后的步骤是灰色的，并且只显示其描述，不会显示这些步骤的动作和其他标记。Eclipse备忘单用这种机制提醒用户这些步骤目前还不能进行。

Eclipse备忘单还允许添加子项。在“设置元素属性”中，创建两个子步骤，如图18-7所示。

```
<item title="设置元素属性">
```

```
<description>在向导对话框中为将要创建的
地址元素设置属性。地址元素的属性包含
两部分，一部分是“名称”，一部分是“类
别”。地址本插件为地址元素设置了9个类别，
需要在这九个类别中选择一类。
</description>
```

```
<subitem label="创建新的子项1"/>
```

```
<subitem label="创建新的子项2"/>
```

```
</item>
```

subitem标签
用来创建新的
子项

子项必须等于或多于两个，否则就会报错
(一个子项也没有必要作为子项)

设置元素属性

在向导对话框中为将要创建的地址元素设置属性。地址元素的属性包含两部分，一部分是“名称”，一部分是“类别”。地址本插件为地址元素设置了9个类别，需要在这九个类别中选择一类。

创建新的子项1

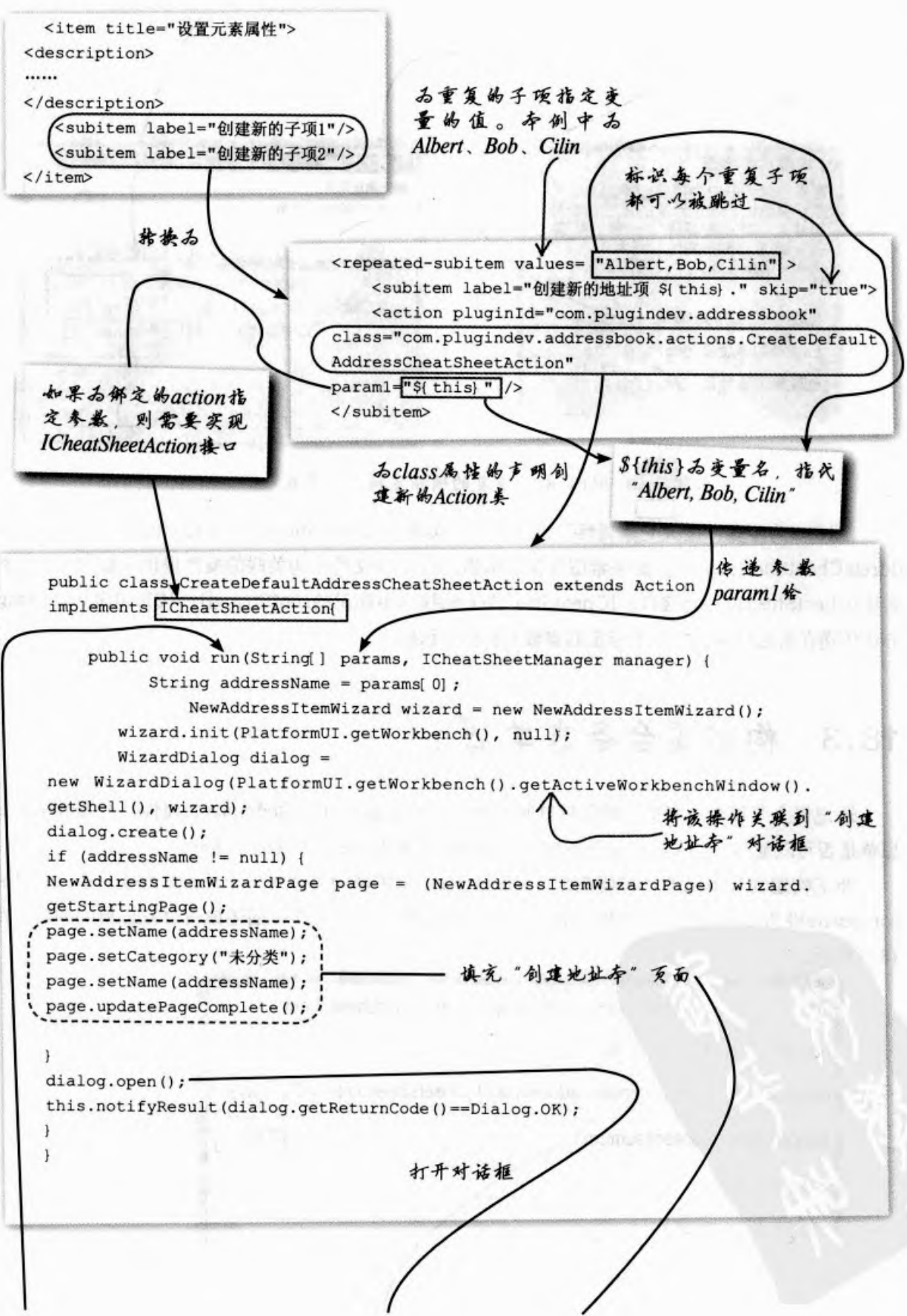
创建新的子项2

单击该标记表示已
经完成子步骤



图18-7 为“设置元素属性”创建子项

子项有一个类似于标题的简短标签，但是没有那么长的描述。子项仍然可以将 skip 属性设置为 true 跳过子项。与项不同，子项不必执行一个严格的操作序列，而是可以按照任何次序执行。更为神奇的是，如果子项所进行的操作相似，那么可以为重复的子项指定变量，通过赋给变量不同的值来表示不同的子项。将图18-7所示的编写子项的方法改进，并且为这些重复的子项绑定一个相同的操作，如图18-8所示。



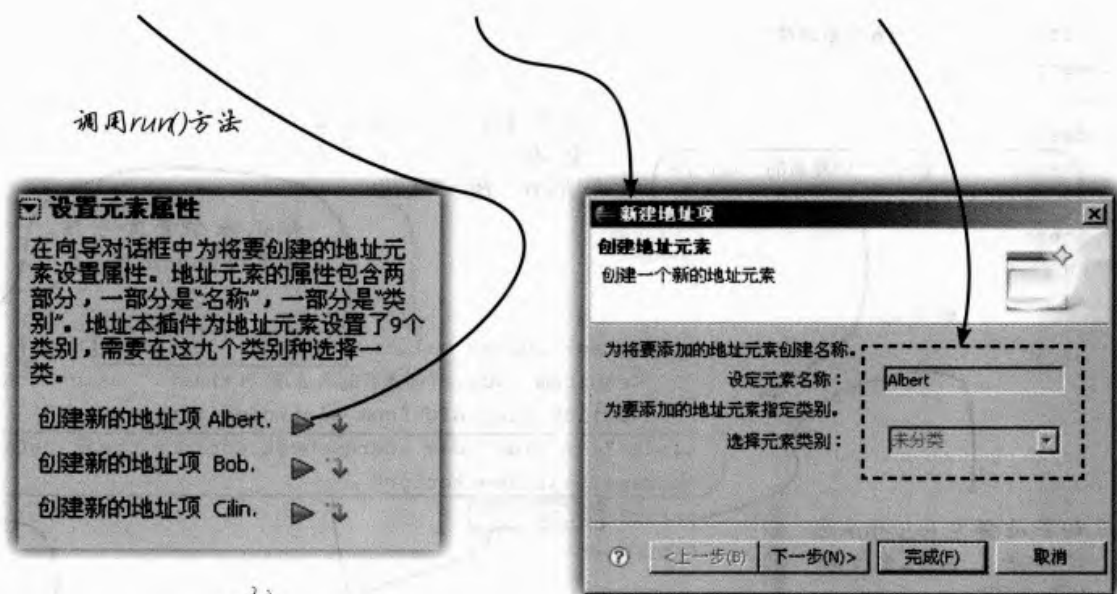


图18-8 为重复的项指定操作和参数

图18-8为步骤“设置元素属性”绑定了一个实现了ICheatSheetAction接口的类CreateDefaultAddressCheatSheetAction。如果希望在备忘单描述的.xml文件中为关联的操作提供参数的话，必须要实现ICheatSheetAction接口。ICheatSheetAction接口中提供的run(String[], ICheatSheetManager)方法传递在备忘单描述文件中指定的参数并且执行该操作。

18.3 构建复合备忘单

还记得上节那个故意被忽略的属性吗？对了，就是备忘单中的composite属性，该属性标识该备忘单是否可以复合。为了构造复合备忘单，需要设置其为true，来改进对话框。

为了构建复合备忘单，需要在cheatsheetContent扩展中再添加一个cheatsheet元素，并且将其composite设为true。然后，创建contentsum.xml文件，在此文件中描述该符合备忘单的内容。如图18-9所示。

id*:	com.plugindev.addressbook.compoundcheetsheet	
name*:	复合备忘单	
category:	com.plugindev.addressbook.cheatsheet.createCategory	
contentFile:	contentsum.xml	<input type="button" value="浏览..."/>
listener:		
composite:	true	▼

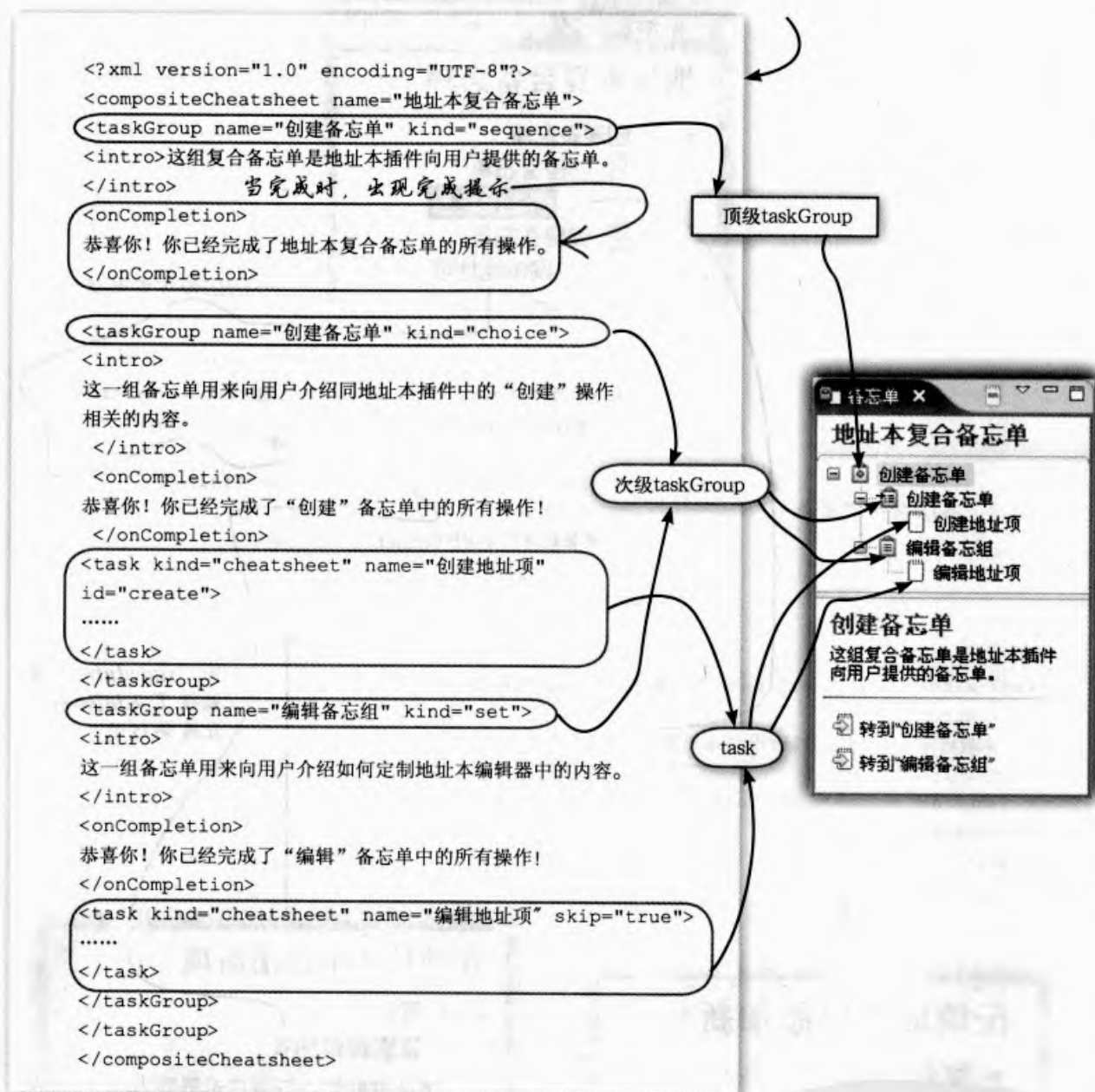


图18-9 构建复合备忘单

在图18-9中，taskGroup的kind属性值有三种，choice, sequence和set，这也是taskGroup可取的全部值。choice意味着用户能够选择任何子任务（组）并完成它；sequence意味着子任务是序列化的，在完成每个任务（组）时必须完成它前面的任务（组）；set表示用户可以用任何顺序随意地完成子任务（组）。

对于task，它可以链接到已经定义的备忘单。将“创建地址项”任务链接到上一节在cheatSheetContent中定义的cheatsheet项中，然后新创建一个名为content2.xml的文件，在其中加入一些描述（在此不再细述创建的过程），将“编辑地址项”任务链接到这个文件中，如图18-10所示。

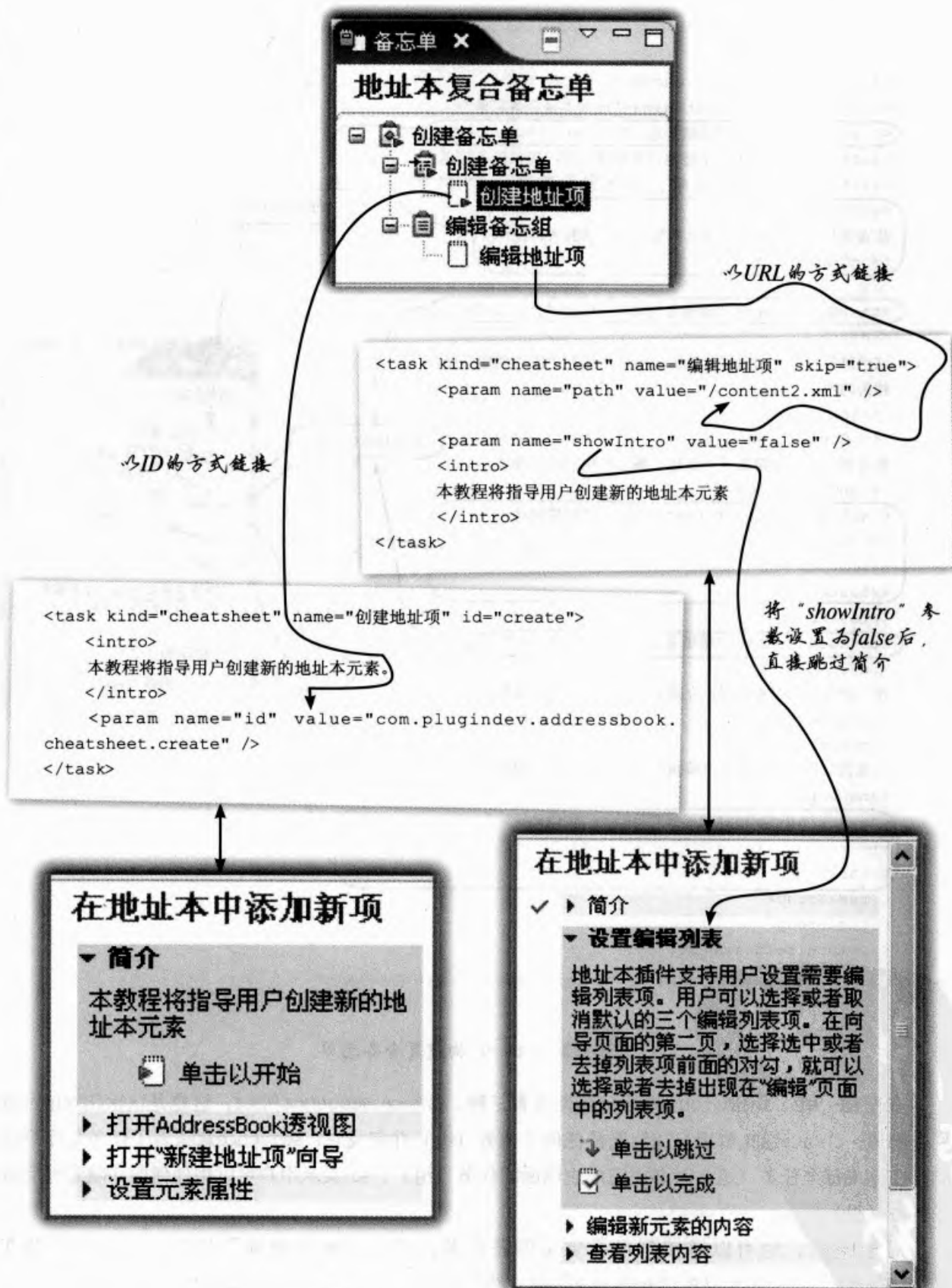


图18-10 链接复合备忘单到单一备忘单

至此，示例程序成功地为地址本插件构建出了复合备忘单。在图18-10中，task标签的param元素用来指代使用的参数。参数类型有三种，在示例程序构建的复合地址本中均使用到了，它们的用法如表18-1所示。

表18-1 任务中可以定义的参数及其用法

名称	用法
id	设置为同任务相关联的备忘单ID
path	备忘单的路径，可以是绝对路径，也可以是相对于该插件的路径
showInfo	表示是否显示备忘单中定义的介绍，如果为false，将直接跳过介绍（复合备忘单也可以为任务定义介绍，这个介绍无法跳过）

18.4 链接备忘单到帮助

备忘单和帮助这两个用户辅助部件，它们有不同的侧重。备忘单能够帮助用户快速上手，而且更加侧重于同用户交互的过程，这种体验往往给用户留下比较深的印象，在不知不觉中掌握操作软件的方法；而帮助侧重于全面的，系统性的介绍，当使用备忘单时遇到不解的问题或想要知道步骤的详细说明或信息时，仍然需要求助于Eclipse帮助。Eclipse提供了链接备忘单到帮助的阶段，从而更方便地定位所需要的帮助话题。

在第17章曾经提到，Eclipse提供了两种方式的帮助，一种是作为帮助文档在“帮助内容”窗口中显示，另外一种是基于上下文的帮助在“帮助”视图中显示。本章将以“添加新地址项”备忘单为例，介绍如何同这两种形式的帮助挂接。

1. 链接到帮助文档

在第17章的主toc文件中提供了quickstart的锚点但并未为其增加任何内容（参见第17章“帮助”），本节中首先为备忘单创建帮助文档，然后为org.eclipse.help.toc扩展中添加一个toc元素，最后创建tocquickstart.xml文件来将帮助文档关联到quickstart锚点中，整个过程如图18-11所示。



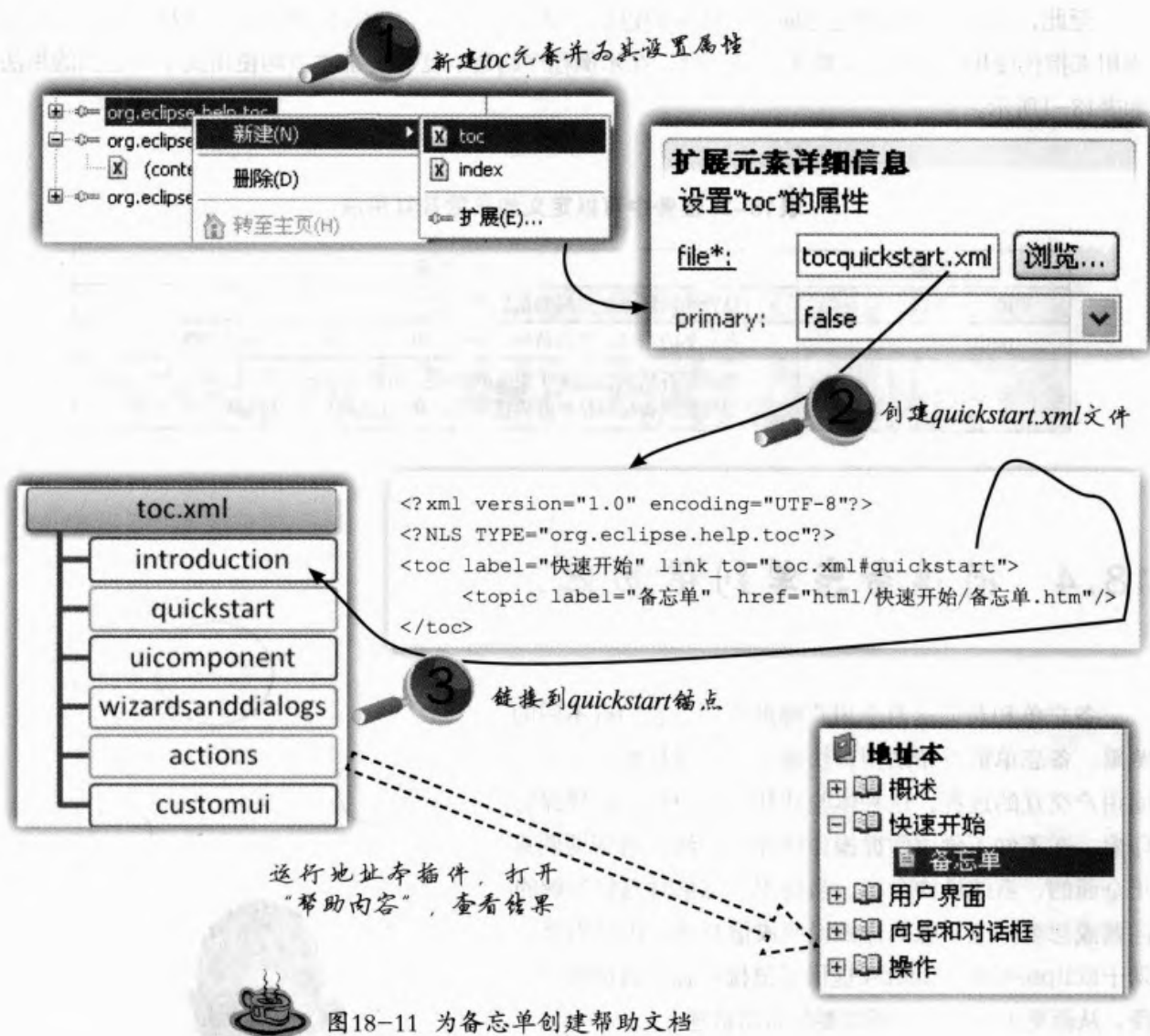


图18-11 为备忘单创建帮助文档

创建完帮助文档后，就可以链接备忘单到帮助文档了。备忘单链接到帮助文档非常简单，只需要在备忘单内容文件的intro元素中添加href属性便可实现该功能，如图18-12所示。



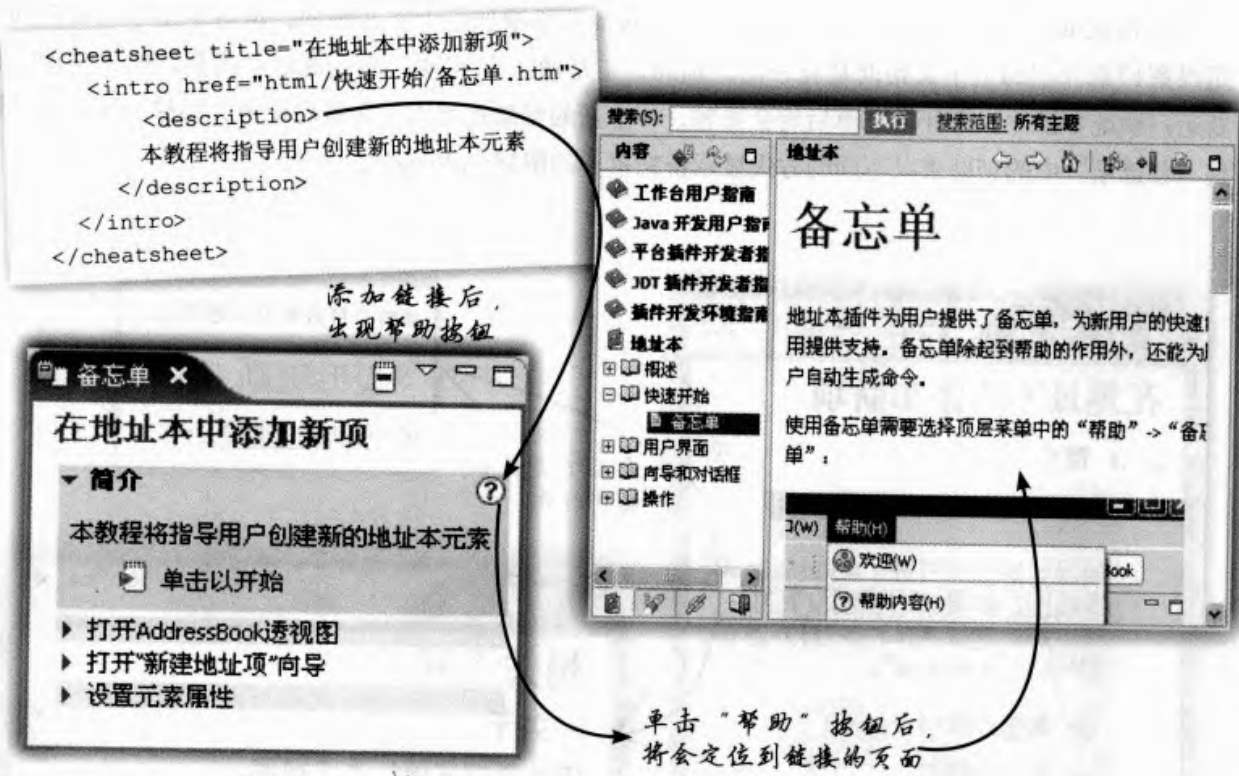


图18-12 添加帮助链接到备忘单

2. 添加上下文帮助

除了可以链接到帮助内容以外，还可以链接到动态帮助中。要连接到动态帮助，仍然需要像在第17章所做的那样，为其声明context_id。

本节将为“新建地址项”备忘单中的第二步“打开AddressBook透视图”添加上下文帮助。首先，在contexts.xml文件夹中声明标识context_id，然后再在contents.xml中为该步骤加入context_id。如图18-13所示。

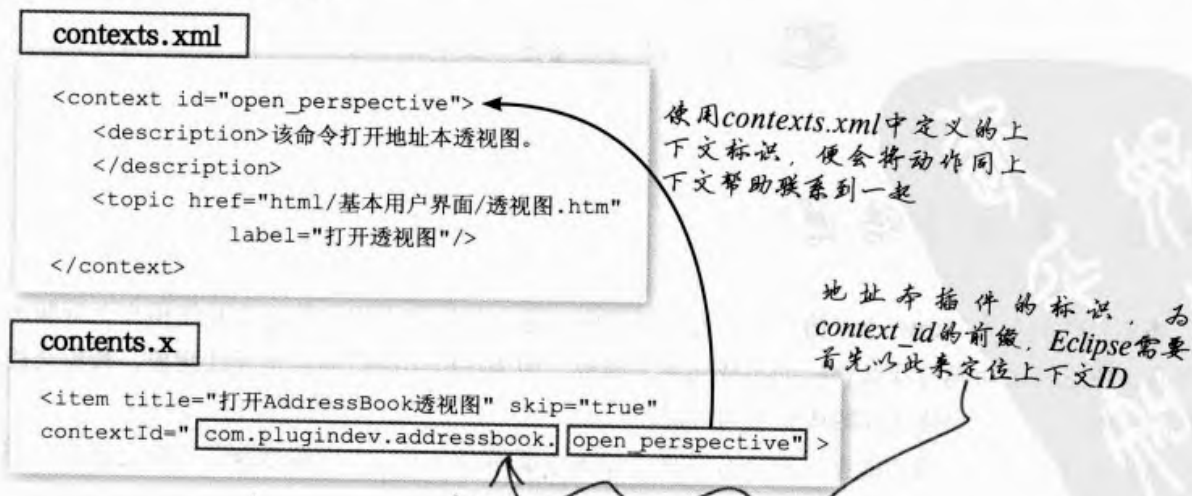


图18-13 链接备忘单和上下文

将备忘单和上下文链接之后，便为Eclipse的备忘单增加了上下文帮助，但这个上下文帮助的使用跟第17章介绍的上下文帮助是有一定区别的，不是通过F1实现，而是通过单击帮助按钮实现。重新运行地址本插件，打开备忘单后将会看到，一个新的帮助按钮出现在备忘单中。同样，可以通过调整“首选项”中的帮助来以不同的方式显示备忘单。如图18-14所示。

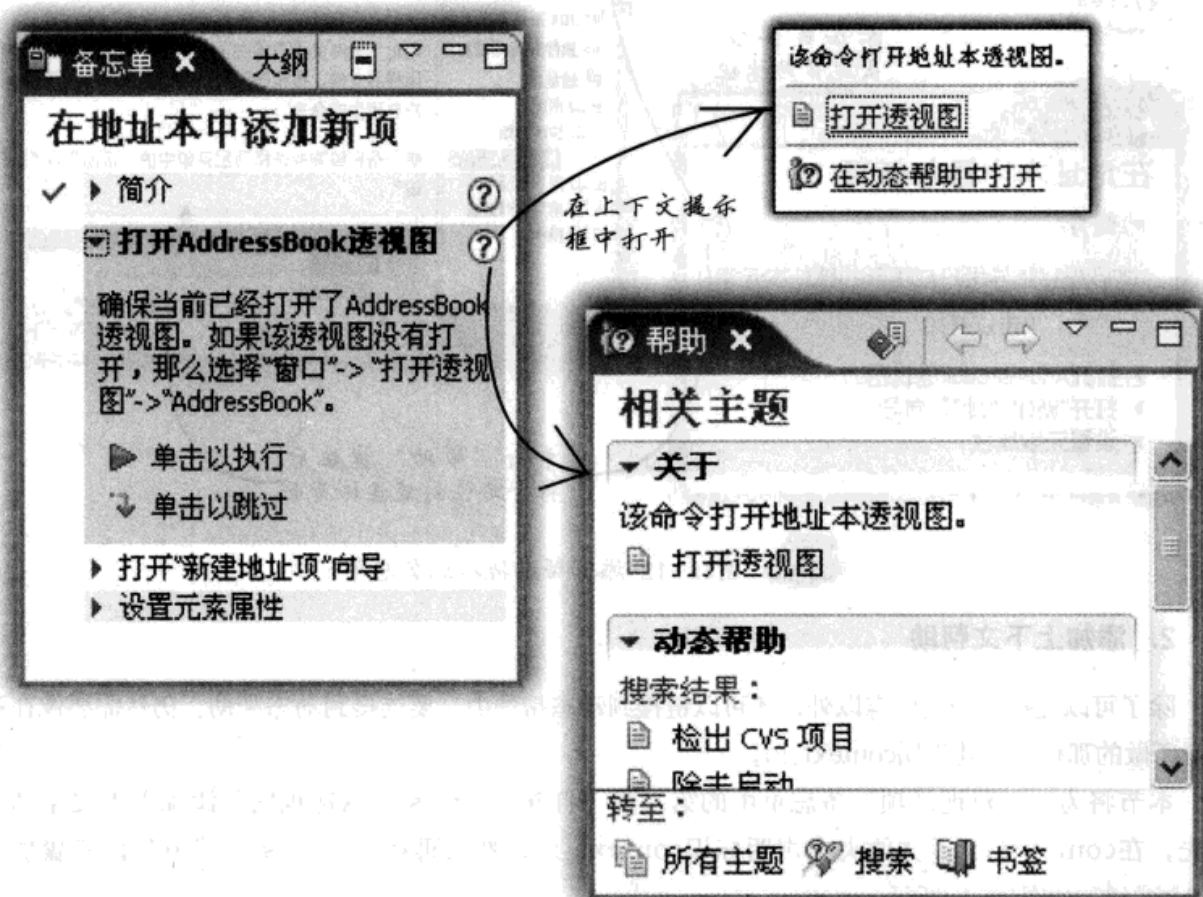


图18-14 运行备忘单上下文帮助

18.5 本章小结

备忘单将教程和 how-to 信息集成进了插件中，这会加快培训用户的速度并让用户更舒服地学习产品。备忘单使功能与 Eclipse 的集成达到了极致，因为在Eclipse以前甚至无法想象，教程会成为产品的一部分，Eclipse使这样的体验成为可能。

备忘单还有许多高级功能在这里没有提及，如备忘单还可以支持Eclipse命令，当开发者在备忘单中加入command命令后，可以使用serialization使其序列化。有关更多内容请参看Eclipse的帮助文档 Composite cheat sheet content file XML format。

本章系统地讨论了Eclipse的备忘单的使用及其创建过程，介绍了为备忘单提供操作的方法。到现在为止，本书几乎已经介绍完了使用插件开发的各种类型的部件，后续的章节将陆续介绍一些更高级的话题。

我们为成功干杯!



拉开崭新的学习帷幕

第19章 插件开发高级内容

本书的第二部分从UI的角度介绍了插件开发的技术，然而要开发一个好的插件产品，只掌握这些是不够的，还需要对Eclipse的插件体系结构有更深入的了解，这包括如何设计插件以保持灵活性、安全性与扩展性，使插件支持国际化等，本章对这些内容进行了概要的介绍。通过本章的学习，读者将能够通过开发自定义的扩展点来增强插件功能，了解如何开发生态的、支持国际化的插件，可以应用段和功能部件组织所开发出的插件，并利用更新站点发布它们。

本章内容包括：

- ★自定义扩展点。
- ★动态插件。
- ★插件的国际化支持。
- ★使用功能部件。



进入第19章

19.1 自定义扩展点

在前面的学习过程中，已经多次基于org.eclipse.ui插件中的扩展点开发过扩展，如开发视图时需要一个基于扩展点org.eclipse.ui.views的扩展，而编辑器则需要基于扩展点org.eclipse.ui.editors。当某个插件需要使用其他插件的服务时，需要在目标插件的扩展点上编写扩展，而当某个插件需要为其他插件提供服务时，就需要编写自己的扩展点。本节将介绍如何在插件中自定义扩展点，以及如何管理和使用基于该扩展点的扩展。

19.1.1 扩展点基础知识

本节中将介绍一些扩展点的基础知识，这包括扩展点定义的组成、扩展点设计实例和一些通用的扩展点设计规则等，这些内容是设计自定义扩展点的基础。

19.1.1.1 扩展点定义的组成

一个扩展点的定义由以下两部分组成。

★plugin.xml文件中扩展点的声明；

★扩展点Schema文件（Schema文件的路径会在声明中指定）。

下面从插件org.eclipse.ui中摘选出了actionSets扩展点的声明作为例子，代码如下所示。

```
<extension-point
    id="actionSets"
    name="%ExtPoint.actionSets"
    schema="schema/actionSets.exsd"/>
```



在plugin.xml中，扩展点用<extension-point>标签来声明，这个标签必须包含id,name,schema三个属性，具体说明如下所示。

id是扩展点的唯一标识，类似于数据库的主键，因而在同一个插件的plugin文件内不允许出现重复。一个扩展点的全局id由插件id加扩展点的id组成，例如，扩展点actionSets的全局名为 org.eclipse.ui.actionSets，这样其他插件在创建基于actionSets扩展点的扩展时，便可以用这个id来指定org.eclipse.ui插件中的actionSets扩展点。

name是对扩展点的一个概要描述。注意，例子中name属性的值以“%”开始，这代表该字符串已被外部化，详见第19.2.3节“外部化plugin.xml中的字符串”。

schema指定了一个扩展名是exsd（Eclipse XML Schema Description）的文件，它对实现这个扩展点的扩展的格式做了定义（如果读者对XML和XSD的知识尚不了解，请参阅相关书籍，这里不再详述）。以下是actionSets扩展点的schema文件actionSets.exsd的一部分（在eclipse目录下搜索“actionSets.exsd”可以找到这一文件），图19-1用图形表示了它的结构，以及该结构与actionSets扩展的对应。

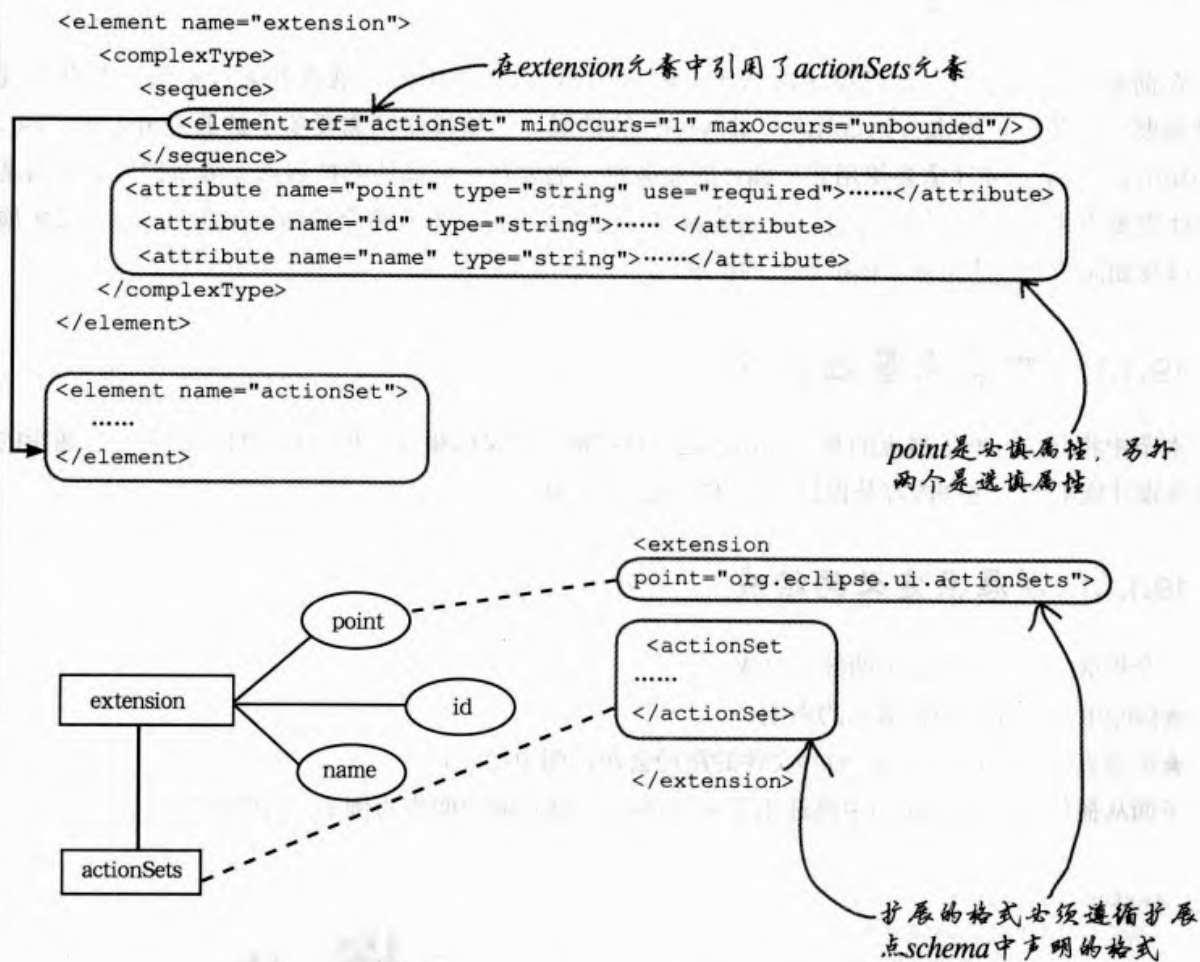
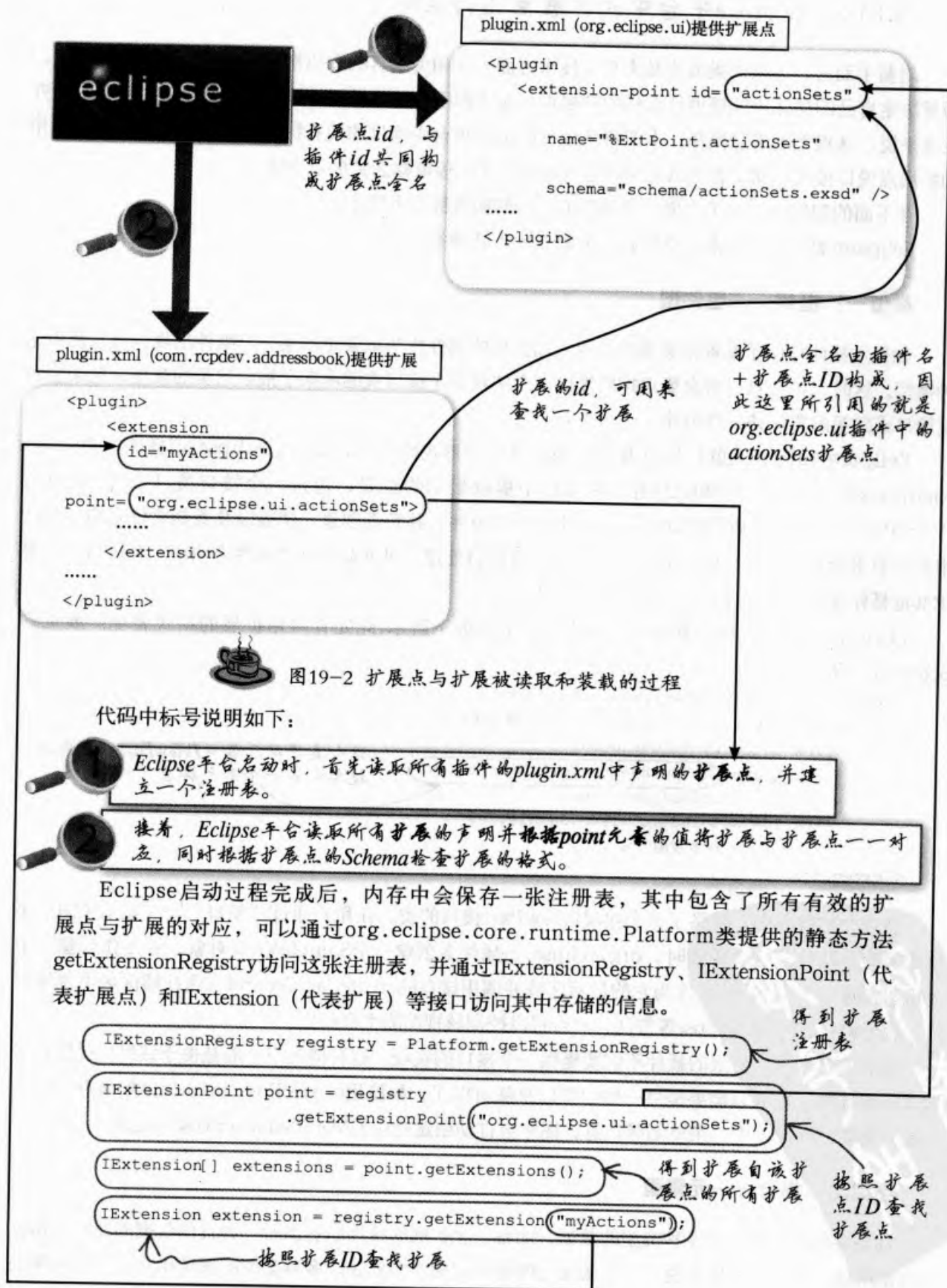


图19-1 actionSets扩展点的Schema



由图19-1可见，扩展点的Schema文件必须以“extension”为根元素，有一个必填属性point和两个可选属性id和name，开发者不能为extension元素添加其他属性，但是可以自己定义extension元素的子元素，如本例中的actionSet元素。在该子元素中，Schema文件会指明扩展点所需的信息及格式，当其他插件扩展该扩展点时，就需要按照这个格式提供信息。

Eclipse平台启动时，会对所有插件的扩展点与扩展进行验证和加载，图19-2仍以actionSets扩展点为例演示了扩展点与扩展的读取和装载过程。



19.1.1.2 Eclipse平台中的扩展点设计实例

对新手而言,最大的挑战不是来自于技术方面——相关内容的介绍都可以在本节稍后部分找到,而是决定自己的插件应该提供什么样的扩展点。对于刚刚开始熟悉插件体系,缺乏足够实践经验的开发者来说,这点尤为难以抉择。本节将从分析Eclipse平台中提供的现有扩展点实例入手,总结出常用的扩展点设计模式,初学者可以从学习这些实例入手,逐渐熟悉扩展点设计。

在下面的叙述中,为了方便,将提供扩展点的插件称为主插件。

Eclipse中繁多的扩展点,总结起来主要有如下几种类型。

类型一: 框架——组件型

如果主插件提供了某种框架类的实现,而扩展的插件作为框架中的某一个组件出现,可以使用这种类型。这时,扩展点通常会要求扩展提供一个实现某个接口或继承某个抽象父类的类型,框架会通过接口或父类的方法访问该组件。

Eclipse中和UI相关的扩展点有很多都是以这种形式出现的,如org.eclipse.ui插件中的views和editors等。平台为这些GUI组件在窗口中提供显示的位置,通过一个接口类(vIEWS对应的IViewPart, editors对应的IEditorPart)访问这些组件,并负责创建、销毁以及重新定位它们,这些组件则负责绘制自己窗口内部的内容并进行业务逻辑处理。另外如果某个插件声明了一个接口,并要求其他插件提供一个实现,也可以使用这种模式。

以views扩展点为例,扩展点的Schema示例如下所示(此处直接用扩展的格式表现扩展点的Schema,下同)。

```
<extension point="org.eclipse.ui.views">
  <category id="plugindemo" name="样本类别"/>
  <view category="plugindemo"
    class="plugindemo.views.SampleView"
    icon="icons/sample.gif"
    id="plugindemo.views.SampleView"
    name="样本视图"/>
</extension>
```

这个类实现了接口IViewPart,它作为框架中的一个单元存在

在这个扩展点中,提供了一个实现IViewPart接口的类。在用户通过“窗口”→“显示视图”菜单或其他方法打开这个视图时,org.eclipse.ui插件会创建一个SampleView对象,在工作台窗口中为视图创建一个Composite作为它的显示区域并调用接口的createPartControl方法在该区域中创建控件。通过接口的init和dispose等方法,平台可以控制该视图的生命周期。

也有的情况下,扩展的插件不需要提供一个接口的实现,这种情况下一般是由于该组件已经有了默认的实现类,扩展只需要提供一些配置信息就可以了。如使用org.eclipse.ui.menus扩展点时,用户就只需要配置菜单ID、菜单名等信息,插件会自动创建对应的MenuManager对象来处理。

类型二: 集合——元素型

如果主插件需要处理多种类型的数据,而每一个扩展的插件则提供对一种具体数据的处理逻辑,可以使用这一类型。举例来说,一个图像处理程序的插件需要处理多种类型的图像格式,这时图像处

理程序可以将具体的图像文件格式处理通过扩展点开放给其他插件，而自身只处理统一后的图像信息，当新的图像格式出现时，只需要为主插件添加一个扩展就可以支持它，如图19-3所示。

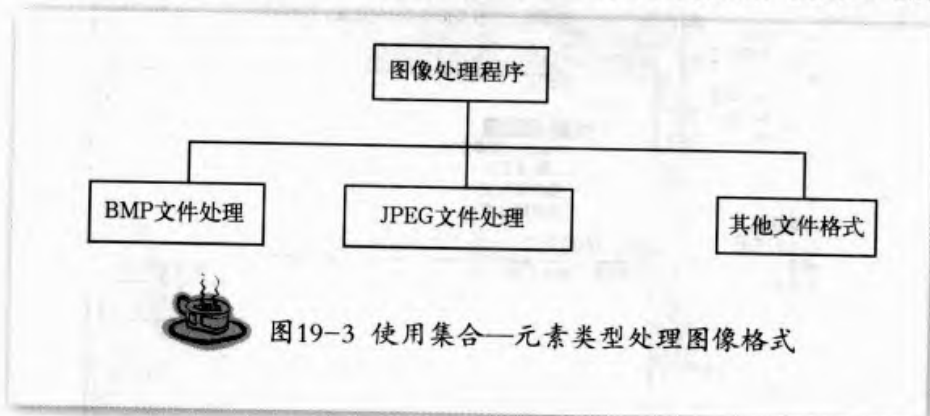


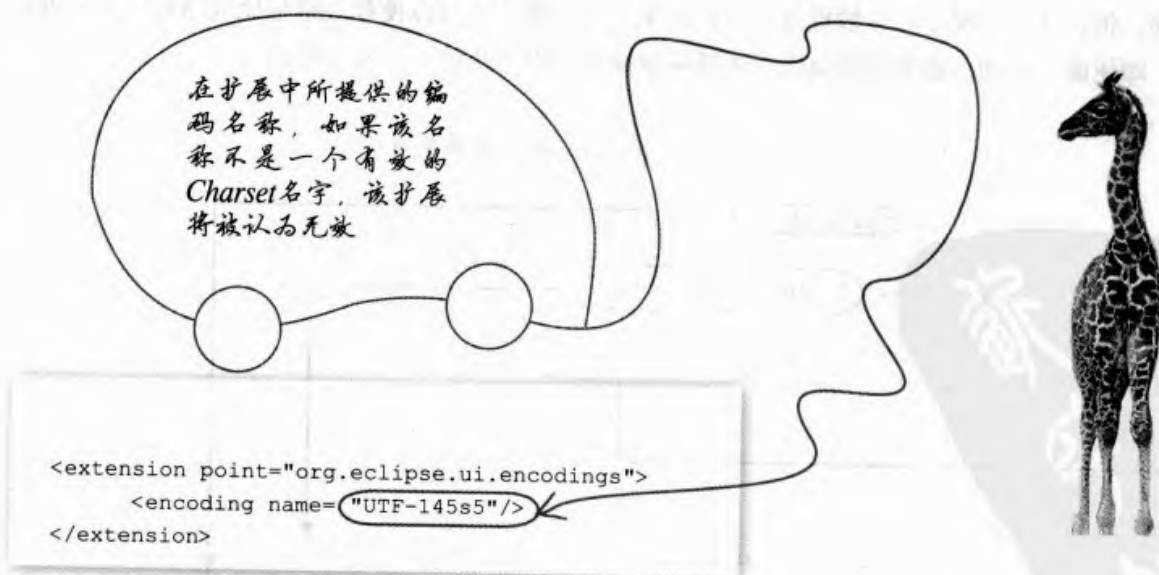
图19-3 使用集合—元素类型处理图像格式

这种类型的扩展点可以只要求扩展提供一个名称，也可以要求其通过接口提供对应的处理程序。如图19-3中所示的图形处理程序就会要求扩展的插件实现一个用于读取和存储文件的接口。

另外，如果主插件提供了功能的多种可行性选择，需要使用它的插件通过扩展来选择使用哪些功能，也可以使用这种类型。这时由于处理逻辑已经全部包含在主插件中，在扩展中只需要提供一个选择，而不需要额外提供处理程序。

属于这种类型的扩展点有org.eclipse.ui.encodings和org.eclipse.core.runtime.contentTypes等，前者用来声明用户在该插件中存储文件时可以选择的字符集编码格式，提供扩展时用户需要一个编码的名字，主插件会使用所提供的名字从Charset类型中取得对应的Charset对象，如果插件使用了该扩展，在存储文件时，用户只能从在该扩展中注册过的字符编码中选择要使用的编码。

下面是一个encodings扩展点的实例。



后者则用于配置文件类型与内容编码的关联，在“首选项”对话框中，选择“常规”→“内容类型”可以看到当前平台中已经配置的文件类型与内容编码的关联情况，如图19-4所示，除手动修改这个页面的内容外，也可以使用扩展点contentTypes向这个页面中添加关联。

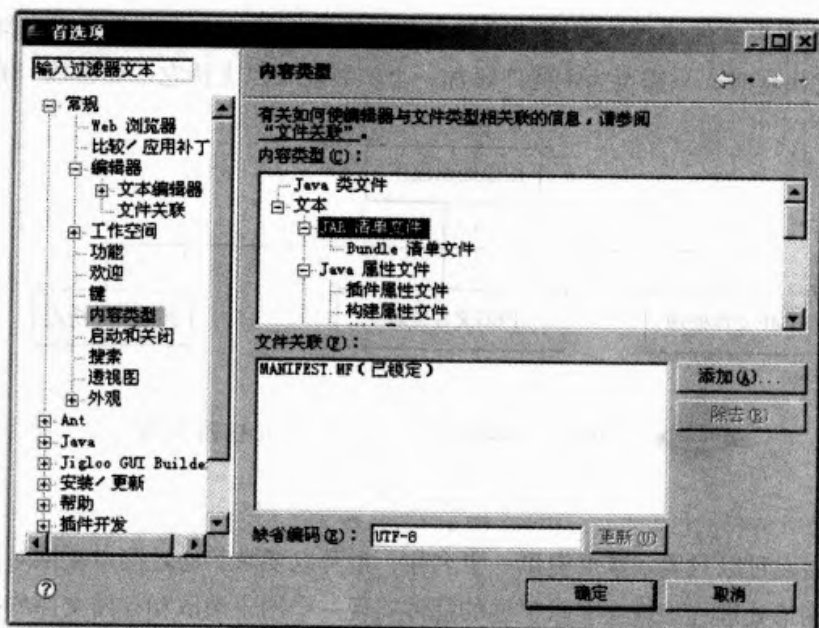


图19-4 内容类型的首选项页面

类型三：用扩展点代替代码

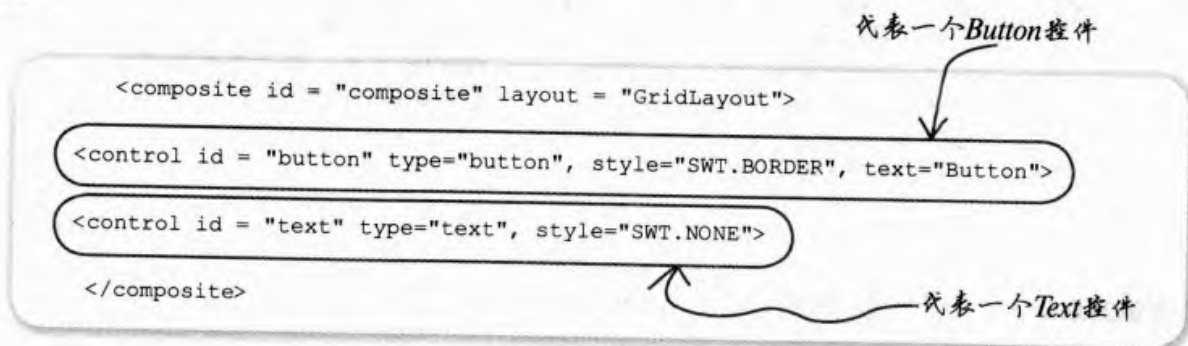
这一类型的扩展点所提供的功能，通常既可以通过配置扩展，又可以通过使用API直接编写代码完成。如org.eclipse.ui中的actionSets中的menu部分就可以归为这一类，在为RCP程序配置菜单时，开发者既可以使用扩展点生成菜单，也可以通过新建一个MenuManager对象来创建菜单，两者效果完全相同，但在这种情况下，一般推荐通过扩展来配置功能，这可以使程序结构比较清晰，易于维护和修改，即使做了改动，也不需要每次都重新编译插件代码。两种方式如下所示。

```
<extension
  point="org.eclipse.ui.actionSets"> 使用扩展实现
  <actionSet .....>
    <menu id="sampleMenu">
      label="Sample &Menu">
        <separator name="sampleGroup" />
      </menu>
    .....
  </extension>
```

```
MenuManager sampleMenu = new MenuManager("Sample
&Menu", "sampleMenu");
sampleMenu.add(new Separator("sampleGroup"));
```

使用代码实现

这种类型的应用可以很广泛，如使用配置的方式在界面上生成控件等，下面的一段XML配置演示了一种可能的实现，当程序需要创建控件时，可以通过分析这段配置取得参数，并据此生成控件对象。



类型四：提供资源

这种类型的扩展点声明了一组资源(如图片、网页文件、文字等)。扩展它的插件提供这些资源内容。拥有扩展点的插件会负责在适当时刻读取并显示它们。

属于这种类型的扩展点有org.eclipse.core.runtime.products和org.eclipse.help.toc等。前者用来声明一个产品配置，产品配置主要用在RCP程序中，它包含了一系列被称为“装饰”的图标和文字，使用产品配置可以设置程序名称（显示在程序窗口标题栏的文字）、程序图标、“关于”对话框中的图片和文字等。本书在第20章中讲述RCP程序开发的内容时，会提到如何使用这个扩展点来装饰程序。

而后者则用于在Eclipse的帮助系统中添加自定义内容，Eclipse的帮助系统，如图19-5所示，由基于XML描述的树型结构（Table Of Content，TOC）和XHTML文件两部分组成，使用toc扩展点，可以将自定义的toc文件和XHTML文件添加到Eclipse的帮助系统中，以为自己的插件创建帮助文档和目录。

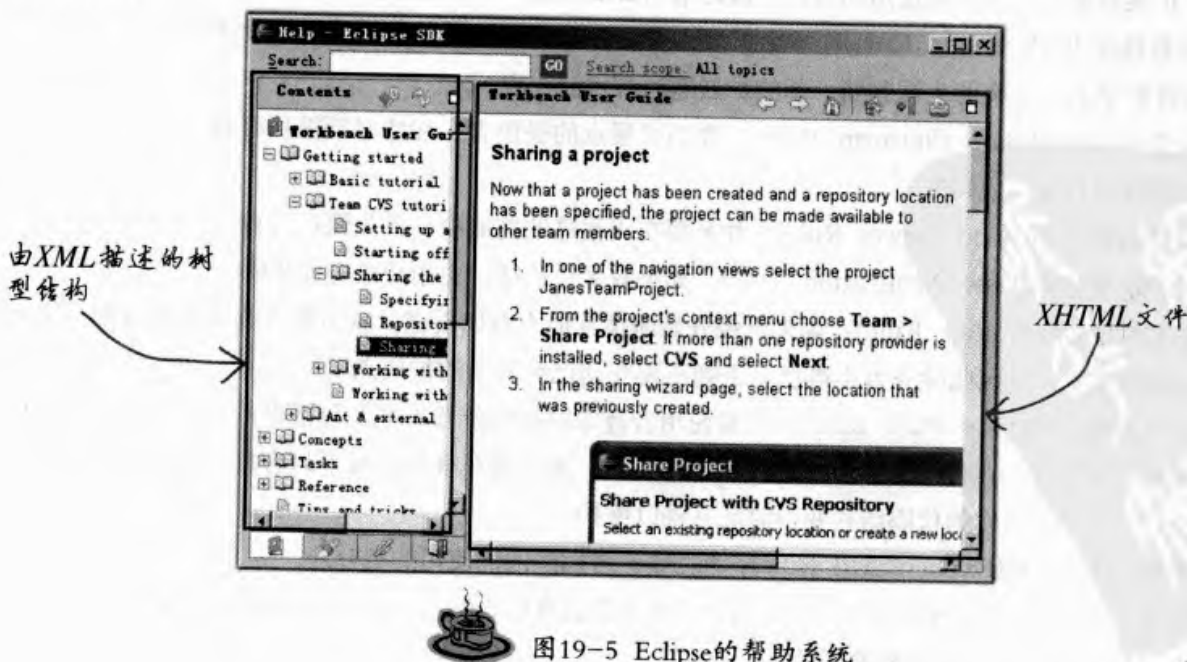


图19-5 Eclipse的帮助系统

下面是使用toc扩展点的一个示例，开发者需要在扩展中提供一系列的XML文件。

```
<extension point="org.eclipse.help.toc">
  <toc file="toc.xml" primary="true"/>
  <toc file="tocconcepts.xml"/>
  <toc file="tocgettingstarted.xml"/>
  <toc file="tocreference.xml"/>
  <toc file="tocsamples.xml"/>
  <toc file="toctasks.xml"/>
</extension>
```

这些XML文件中指定了对应的XHTML文件，因此在扩展中不需要提供XHTML文件

在第20章讲述RCP程序的开发时，将更深入地讲述toc扩展点的使用。

19.1.1.3 设计扩展点的规则

作为一个扩展点的提供者，最好能够了解以下通用法则。这些规则选自Eclipse社区，是广大开发者们在长期实践中总结出来的经验。

邀请法则(Invitation Rule): 尽可能地邀请别人为你的作品做出贡献。只有使用的人多了，才能及时发现和更改插件中存在或潜在的问题；而且开发扩展点的本意就是为了让别人使用，打打广告也很有必要。

懒加载法则(Lazy Loading Rule): 只有在真正需要的时候才加载插件。不要试图在插件启动时就将所有扩展的插件加载上来，这可以为用户节省内存空间。另外也不应该依赖于“加载几个类不会占用太多空间”一类的推测。因为开发者永远不可能完全清楚将要加载进来的、属于别人的插件会做些什么事情。

明确扩展法则(Explicit Extension Rule): 明确说明插件的什么地方可供扩展，以及扩展它们的结果。扩展点是为了供别人使用而设计，因此书写必要的说明文字很有意义。

发散性法则(Diversity Rule): 一个扩展点接纳多个扩展。不要在扩展点的设计中加入“只有一个插件扩展我”之类的主观判断，新的扩展随时都可能出现。

安全平台法则(Safe Platform Rule): 作为扩展点的提供者，插件必须保护好自己，不要让扩展者的误操作对自身造成损失。

良好防御法则(Good Fences Rule): 如果要交出程序的控制权，首先保护好自己。

用户决定法则(User Arbitration Rule): 如果有多个选择，由用户决定使用哪一个。

稳定性法则(Stability Rule): 如果已经开始邀请其他人做出贡献，就不要再改变游戏规则。大家的时间都是宝贵的，因此除非万不得已，否则不要变动已经发布的扩展点Schema。

公平竞争法则(Fair Play Rule): 所有使用者遵守同样的游戏规则，包括自己。如果声明了扩展点来完成某项工作，在自己的插件中也应该使用它们，而不是在插件中留下只有自己知道的后门来绕开扩展点，这只会使你的代码结构变得混乱并难以维护。

明确 API 法则(Explicit API Rule): 将 API 与插件内部使用的类分开。

保守 API 法则(Defensive API Rule): 只暴露自己有信心的 API，但同时也应该做好准备暴露更多的 API，因为使用者会要求这样做。

这些规则中最基本的内容可以总结为以下两点。

- ★设计定位明确，有规范的API的扩展点并为之编写文档。
- ★保护提供扩展点的插件，防止它被未知的扩展非法使用从而导致错误。

了解了这些原则后，可以开始尝试设计自己的扩展点了。

19.1.2 使用扩展点

本节中，将在Address Book插件中添加一个扩展点，通过这个扩展点，开发者可以为插件设置一条文字欢迎信息。

为了显示该条信息，在工作台的工具栏中添加了一个按钮，单击该按钮就会弹出一个包含欢迎信息的对话框，如图19-6所示。



图19-6 通过扩展点为插件提供欢迎信息

19.1.2.1 创建扩展点

第一步，需要在插件的plugin.xml文件中添加一个扩展点声明。用插件清单编辑器打开文件，切换到“扩展点”选项卡，并单击“添加”按钮新建一个扩展点，新建的扩展点会出现在选项卡的列表里面，如图19-7所示。



创建了声明后，如果在“新建扩展点”的对话框中选中了“完成时编辑扩展点模式”一项，平台会自动打开生成的Schema文件，否则，也可以右键单击扩展点，选择“打开模式”来打开它。Schema的编辑器提供了三个选项卡，“概述”选项卡中包含了一些扩展点描述、版权等扩展点信息，这些都是描述性的内容，对扩展点的开发没有影响；在“定义”选项卡中可以以图形化的方式编辑Schema，也是开发者完成主要工作的地方；最后一个选项卡是一个文本编辑器，有经验的开发者可以在这里手动编辑Schema文件。

打开“定义”选项卡，可以看到自动生成的extension根元素。在讲述扩展点格式时提到过，用户声明的Schema必须以这个元素为根，它的格式也是固定的，不能向其中添加属性，因此选中这个节点时，“新建属性”的按钮会被禁用，用户只能向其中添加自定义的元素。

下面就来建立自己的扩展点Schema。首先单击“新建元素”新建一个自定义元素并为它命名。用户可以为自定义元素的属性指定各种限制，包括类型(boolean、string、java等)，是否为必填项(如果是必填，在创建扩展时一定要提供这个属性值)等，如果类型选成java，代表属性的值必须是一个类名，这时还可以指定这个类需要继承的父类或实现的接口。

在示例中，建立了一个名为impl的元素，并为它声明了name和class两个属性。其中class属性被

指定为必须是一个实现了com.plugindev.addressbook.IMessageProvider接口的类，通过这个接口，扩展可以向扩展点提供欢迎消息内容。如图19-8所示。

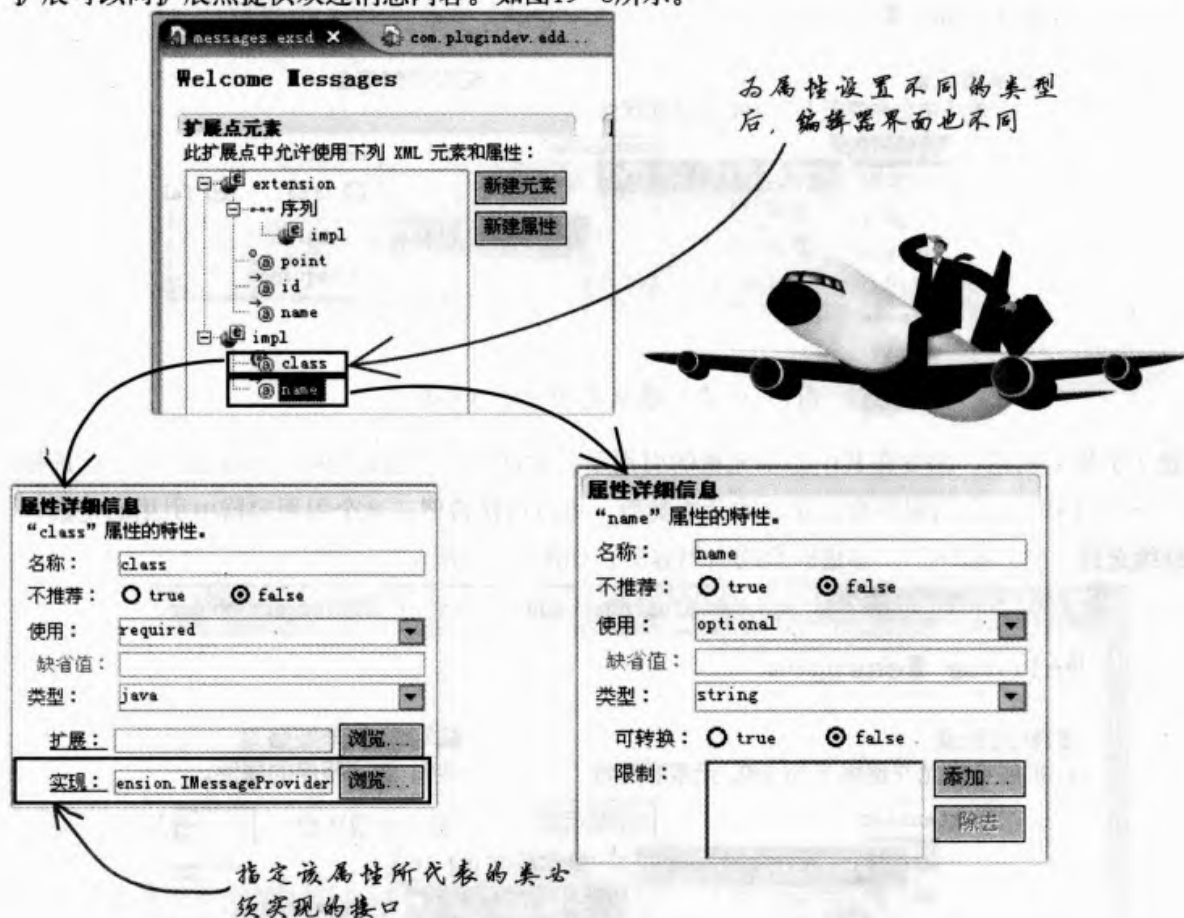


图19-8 为扩展点添加元素声明

IMessageProvider的代码如下所示，它的两个方法分别为所显示的对话框提供了标题和内容文字。

```
public interface IMessageProvider {
    public String getTitle();
    public String getMessage();
}
```

当其他插件实现这个扩展时，会引用到该接口，因此需要将它加入到插件的导出列表中。在“运行时”扩展卡中，将该接口所在的包“com.plugindev.addressbook.extension”加入到“已导出的包”列表中。

创建了合适的元素后，最后一步是将其引用关联到extension根元素上，根据XSD的规范，首先要创建一个引用容器，右键单击extension元素，选择“新建”→“排字程序”。子菜单中有choice和sequence两个选项。开发者可以根据需要在这两种引用容器中选择，如图19-9所示，创建了引用容器后，可以在右边编辑它的属性，如出现次数等，此处将impl元素的最小出现次数和最大出现次数均规定为1，这也就意味着一个扩展中有且只能有一个impl元素。

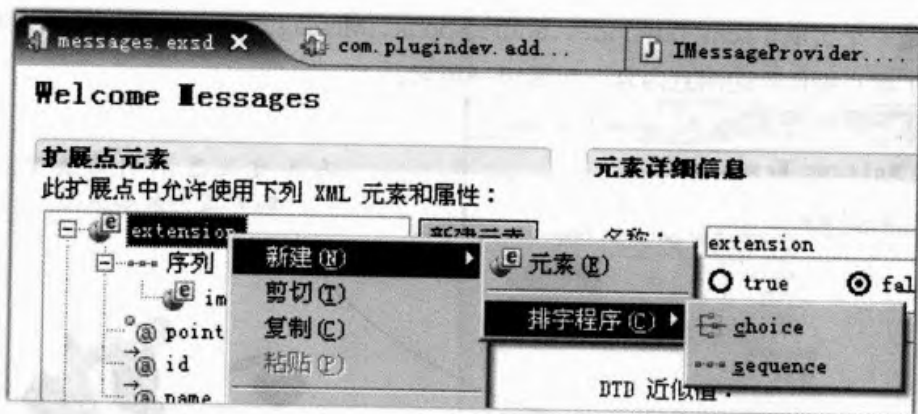


图19-9 在扩展点声明中添加序列

创建了引用容器后，需要在其中添加元素的引用了。右键单击所创建的sequence或choice，选择“新建”→“引用”，会出现所有已建立的元素类型，可以选择希望在这个引用容器中引用的元素，也可以继续选择“排序程序”，创建嵌套的引用容器，如图19-10所示。

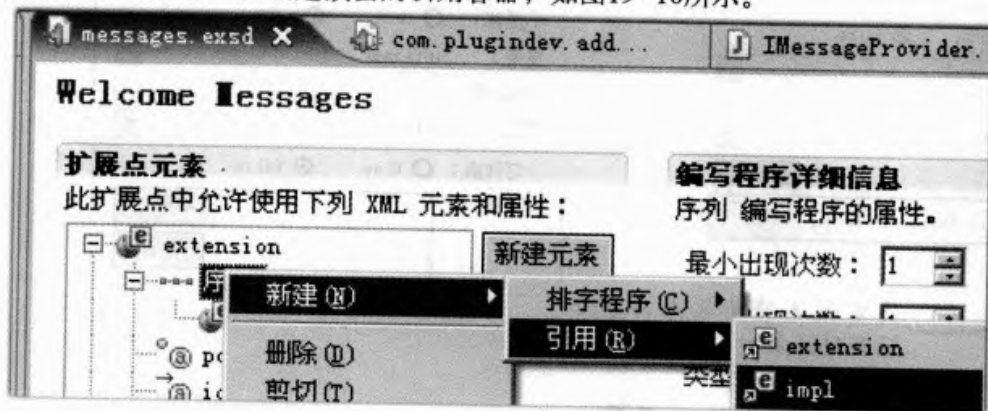


图19-10 编辑扩展点声明

将impl元素添加到sequence中后，就完成了扩展点结构的声明，所创建的Schema结构如图19-11所示。

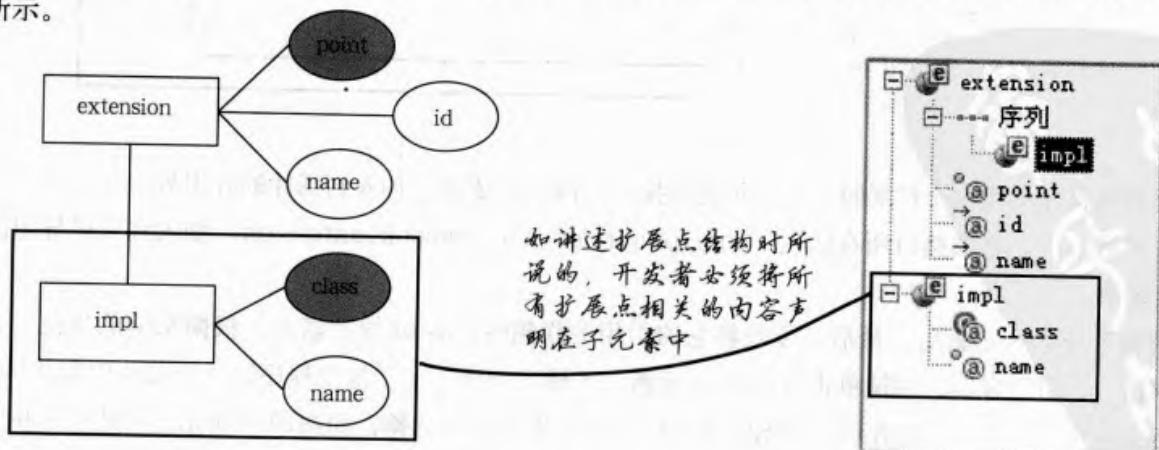


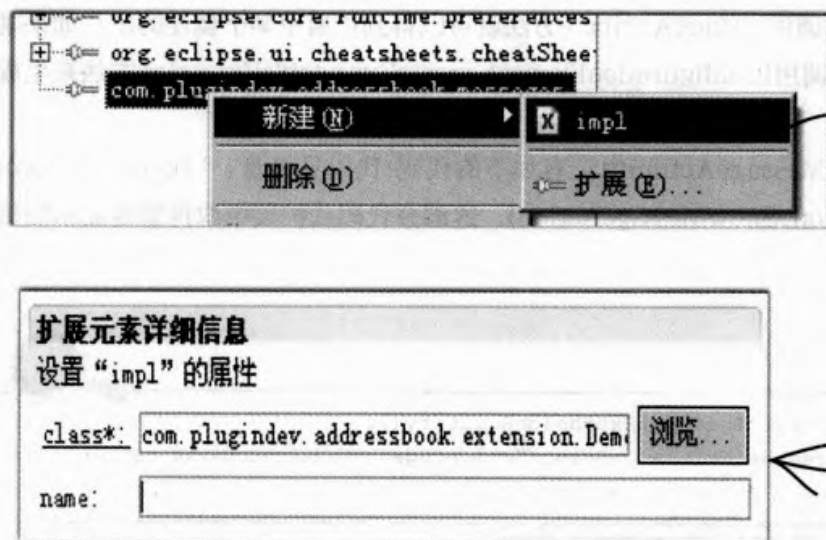
图19-11 创建的messages扩展点结构

现在，可以基于这个扩展点添加扩展了。

19.1.2.2 为扩展点增加一个扩展实例

在自定义的扩展点上添加扩展和使用Eclipse平台提供的扩展点没有什么区别，为了方便起见，本示例将不再创建一个新的插件，而是将该扩展直接放在地址本插件里面。

打开plugin.xml，在“扩展”选项卡中单击“添加”按钮，选择上一节中创建的扩展点“com.plugindev.addressbook.messages”，将它添加到扩展列表中，然后右键单击添加的扩展，选择“新建”，上一节中所创建的impl元素会出现在右键菜单中，可以选择它来新建一个impl元素，如图19-12所示。



创建了impl元素后，需要配置它的两个属性：class和name



图19-12 从扩展点创建并配置一个扩展

由于在扩展点中将class属性指定为一个实现了IMessageProvider接口的类。现在需要创建一个这样的类型。新建一个Java类并取名为DemoProvider，其实现代码如下所示。

```
public class DemoProvider implements IMessageProvider {
    public String getMessage() {
        return "这是扩展提供的欢迎信息";
    }
    public String getTitle() {
        return "欢迎信息";
    }
}
```



19.1.2.3 在扩展点中调用扩展

现在扩展点与扩展都已经开发完成了，最后一步工作是在提供扩展点的插件中编写调用扩展的代码。

下面一段代码可以用来得到基于messages扩展点的扩展，它访问了Eclipse的插件注册表以获得这些信息。

```
IExtensionRegistry reg = Platform.getExtensionRegistry();  
IExtensionPoint ep = reg.getExtensionPoint(  
    "com.plugindev.addressbook.messages");  
IExtension[] extensions = ep.getExtensions();
```

一个IExtension对象就代表了一个扩展。而扩展中声明的、作为extension子元素出现的那些元素，被称为扩展的配置元素(Configuration Element)，通过接口IConfigurationElement可以访问它们，用IExtension.getConfigurationElements可以得到某个扩展中所包含的所有子元素，调用getName方法可以得到该配置元素的名称，而调用getChildren方法可以得到该配置元素的子元素集合，使用这两个方法可以定位一个已知名称的配置元素。

定位到所需的配置元素后，调用它的getAttribute方法就可以得到扩展中某个属性的值，如果某个属性的类型被声明为java，可以调用IConfigurationElement.createExecutableExtension方法来生成一个对象实例。

在负责处理操作的类ShowMessageAction中，有如下的代码(代码见光盘：\book.ch19.com.plugindev.addressbook.extension.ShowMessageAction)，这部分代码从扩展中取得需要显示的信息并创建对话框。

```
.....  
public void run(IAction action) {  
  
    IExtensionRegistry reg = Platform.getExtensionRegistry();  
    IExtensionPoint ep = reg.getExtensionPoint("com.plugindev.addressbook.messages");  
    IExtension[] extensions = ep.getExtensions();  
  
    for (int i = 0; i < extensions.length; i++) {  
  
        IExtension ext = extensions[i];  
  
        IConfigurationElement ce =  
            ext.getConfigurationElements()[0];  
  
        if (!"impl".equals(ce.getName()))  
            return;  
  
        try {  
  
            String name = ce.getAttribute("name");  
            Object obj = ce.createExecutableExtension("class");  
            IMessageProvider provider = (IMessageProvider) obj;  
            MessageDialog.openInformation(window.getShell(), "From " + name  
                + ":", provider.getTitle(), provider.getMessage());  
  
        } catch (CoreException e) {  
            e.printStackTrace();  
        }  
    }  
}
```


代码中标号说明如下:



调用Platform API, 通过扩展点名称获取扩展点, 通过扩展点获取扩展。



该扩展点并不限制有多少插件为其提供扩展, 因此这里用一个循环来处理所有扩展, 对每一个扩展都会显示一个对话框。



在声明扩展点格式时, 已经限制有且只能有一个impl元素出现, 因此这里直接取数组中的第一个元素。



因为该实例结构比较简单, extension元素只有impl一个子元素, 因此这句判断条件事实上是永远为false的, 该处只是演示一下如何使用 IConfigurationElement.getName方法。



从扩展中读取name属性和class属性的值, 并根据class属性由Eclipse平台生成一个IMessageProvider的实现类, 将内容拼起来显示一个对话框。

※ 注意: ※

如果扩展点要求扩展提供一个Java类型的属性, 并需要将其实例化以得到一个Java对象的话, 一般情况下都应该使用IConfigurationElement的createExecutableExtension方法来创建实例, 而不应该通过getAttribute方法得到类名后试图自行创建。这是和Eclipse平台的类读取规则相关的。

在Eclipse平台中, 每个插件都拥有独立的类空间和类读取器, 一般来说, 在插件A中不应该直接通过类名直接读取并访问插件B中的类型, 而必须通过平台提供的途径, 如上述的createExecutableExtension方法来访问其他插件中的类(但如果某个类型在插件B“已导出的包”列表中, 那么其他依赖于B的插件就可以直接引用它了), 这也保证了插件之间只通过扩展点相互交流的松耦合度。



19.2 插件的国际化支持

作为一个商业化软件产品, 对国际化(Internationalization, i18n)的支持是必不可少的。软件的国际化支持是指在开发时合理地设计软件体系结构, 使得同一套软件产品可以适应全球不同国家和地区的用户的使用需求。这些需求包括了软件的界面语言、标点符号、文字/日期显示格式, 以及与文化风俗相关的诸多内容。开发者可以为支持国际化的软件产品动态补充新的支持内容, 而不需要重新编译整个产品。

Eclipse的插件国际化技术基于Java语言所提供的国际化实现, 本节将首先对JDK中的国际化技术进行介绍, 然后将以改造地址本插件为例, 讲述如何为插件产品添加国际化支持。

19.2.1 国际化方法

通常一提到国际化, 最直观的理解就是对多语言的支持, 然而这并不是国际化工作的所有内容。

一个支持国际化的程序需要满足以下条件。

★在不同的国家/地区中，程序可以以当地化的界面运行。当开发者为程序添加对新的地区的支持时，不需要重新编译程序。

★文本内容(如界面标签、提示信息以及帮助文档等)不是硬编码在程序中。它们与主程序分开来存储，在运行时根据系统设置动态读取。

★程序中所使用的日期/时间、汇率、度量衡等的格式都是符合用户所在地区的使用习惯的。

与国际化所对应的概念是本地化(Localization, l10n)，本地化是指将一个软件改造成符合某一地区用户使用习惯的过程。本地化工作主要涉及以下几部分内容。

★文本 主要包括界面上的标签文字、帮助文件等。这通常是本地化工作量最大的一部分。程序员需要为不同的国家提供对应的文本。

★数据格式 如数字、日期、时间、货币、电话号码、邮递地址等的格式。

★地区和个人称谓 度量衡、尊称和个人头衔等。

★多媒体考虑事项 在程序界面中使用的图片、声音等内容需要符合当地人的文化习俗。

本地化与国际化是相互依赖的两个概念。一个程序国际化的程度越高，将这个程序本地化的工作就越简单。

19.2.1.1 使用Locale格式化数字、日期和度量衡

在Java中，使用java.util.Locale类型存储与地域相关的内容，一个Locale对象代表着一个地理上、政治上或文化上独立的区域，在使用JDK中与地域相关的功能时(这些功能通常与国际化有关，如日期、数字等的显示格式)，需要使用Locale对象指明所在的地域。

通常使用以下两个构造函数来创建一个Locale对象。Locale(String language)或Locale(String language, String country)，language参数必须是一个有效的ISO语言代码。ISO语言代码是ISO(International Standard Organization，国际标准化组织)制定的一份语言列表。它包含了世界上所有主流语言，并为每种语言指定了一个由两位小写英文字母构成的代码，如英文的代码是en，中文的代码是zh，而日语的代码则是jp。以下网址列出了一份完整的ISO语言代码表，<http://www.loc.gov/standards/iso639-2/englangn.html>。

与此类似，country参数也必须是一个有效的ISO国家代码。国家代码由两位大写英文字母构成，如美国是US，英国是UK，中国是CN等。完整的ISO国家代码表可以在以下网址找到，<http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html>。

国际化的最主要障碍是语言部分，因此如果只需要按语种区分各地区时，使用第一个构造函数构造Locale就够了；但是考虑到同一个语种的地区仍然可能有文化和习俗差异(如英国英语和美国英语就存在显著差别)，如果需要将地域划分得更细致，可以使用第二个构造函数。

在下文中提到代表某个具体地区的Locale对象时，通常会用形如(语言代码)_(国家代码)的格式代表它。如en_US就代表了美国地区。

下面的代码构造了一个代表中国大陆地区(zh_CN)的Locale。

```
Locale locale = new Locale("zh", "CN");
```


在Locale类中内置了很多常用国家/地区的Locale对象，可以直接使用。如代表中国大陆的Locale.CHINA，代表中国台湾的Locale.TAIWAN等。

数字和日期等内容的存储格式是与地域无关的，但在显示给最终用户时，不同的地区对显示格式要求却有很大区别。在开发支持国际化的软件产品时，这一点是不可忽视的。举一个简单的例子，显示数字时，亚太地区使用逗号作为千位分隔符，点号作为小数分隔符。在这些地区，2,300代表了2300；而部分德语国家的使用习惯却正好相反。在那里，2.300才代表2300。如果一个全球化的软件产品不注意这些细节，势必引起很多纠纷。

JDK提供了java.text.NumberFormat类用于处理这些问题。NumberFormat类内置了很多国家/地区的数字显示格式。指定一个Locale，就可以将数字转化成对应地域格式的字符串输出。下面的代码演示了如何使用NumberFormat格式化数字。

```
NumberFormat chinaFormat = NumberFormat.getInstance(Locale.CHINA);
System.out.println(chinaFormat.format(5542.22d));
NumberFormat germanFormat = NumberFormat.getInstance(Locale.GERMAN);
System.out.println(germanFormat.format(5542.22d));
NumberFormat deFormat = NumberFormat.getInstance(
    new Locale("de", "CH"));
System.out.println(deFormat.format(5542.22d));
```

其显示结果分别如下所示。

5, 542. 22
5. 542, 22
5' 542. 22

使用NumberFormat也可以很方便地将数字转化成百分数。这种转化同样支持国际化定制，代码如下所示。

```
NumberFormat usPercentFormat = NumberFormat.getPercentInstance(Locale.US);
System.out.println(usPercentFormat.format(.22d));
```

其显示结果如下所示。

22%

如果软件产品涉及经济内容，除了显示数字外，通常还需要显示对应国家的货币符号。如美元的“\$”，人民币的“¥”等。NumberFormat也可以提供某个Locale对应的货币符号，如下面代码所示。

```
NumberFormat cCFormat = NumberFormat.getCurrencyInstance(Locale.CHINA);
System.out.println(cCFormat.format(12345.67));
NumberFormat cUsFormat = NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(cUsFormat.format(12345.67));
NumberFormat cUkFormat = NumberFormat.getCurrencyInstance(Locale.UK);
System.out.println(cUkFormat.format(12345.67));
NumberFormat cJpFormat = NumberFormat.getCurrencyInstance(Locale.JAPAN);
System.out.println(cJpFormat.format(12345.67));
```


运行结果如下所示。可以注意到NumberFormat在格式化货币时，将货币的精度也考虑了进去。因为日元没有小数，用日元格式格式化小数时会自动四舍五入。

```
¥12,345.67
$12,345.67
£12,345.67
¥12,346
```

java.util.Currency类中还提供了许多与货币相关的信息。要确定一种货币符合常规的小数位数，可以使用 getDefautFractionDigits() 方法，如下面代码所示。

```
Currency c = Currency.getInstance("KRW");
int i = c.getDefautFractionDigits();
```

处理日期数据时，需要使用java.text.DateFormat类。与NumberFormat类相似，DateFormat可以根据用户所选的Locale按照对应地域的日期格式将Date对象格式化成字符串。使用DateFormat之前，首先要指定所用的日期/时间格式，这些格式指示着在格式化过程中应当将一个Date对象中的哪些信息转化到字符串中。DateFormat内置了几个格式。程序员可以使用它们创建DateFormat的实例并调用它的format方法来格式化Date对象如下面代码所示。

```
Date instance = new Date();
```

```
DateFormat format =
DateFormat.getDateInstance(DateFormat.SHORT, Locale.CHINA);
```

调用getDateInstance得到一个DateFormat对象，它只格式化Date对象中的日期信息。也可以调用getTimeInstance或getDateTimeInstance。前者得到的DateFormat只格式化Date对象中的时间信息，而后者则会将日期和时间一起格式化。

```
String formattedDate = format.format(instance);
```

表19-1列出了在英/中文的Locale下，使用DateFormat的内置格式转化同一个Date对象的结果(Date对象代表的日期是2011年10月25日)。

表19-1 DateFormat中关于日期的格式

格式	英文Locale	中文Locale
DateFormat.SHORT	10/25/11	11-10-25
DateFormat.MEDIUM	Oct 25, 2011	2011-10-25
DateFormat.LONG	October 25, 2011	2011年10月25日
DateFormat.FULL	Tuesday, October 25, 2011	2011年10月25日 星期二

对Date对象中时间的转换也是类似的(使用DateFormat.getTimeInstance得到用于格式化时间的DateFormat对象)，如表19-2所示。

表19-2 DateFormat中关于时间的格式

格式	英文Locale	中文Locale
DateFormat.SHORT	3:32 PM	下午3:32
DateFormat.MEDIUM	3:32:00 PM	15:32:00

DateFormat.LONG	3:32:00 PM GMT+08:00	下午03时32分00秒
DateFormat.FULL	3:32:00 PM GMT+08:00	下午03时32分00秒 GMT+08:00

除日期格式外,时区也是国际化软件不可忽略的因素之一。时区具有两个主要的属性。第一个是时区与格林尼治标准时间(GMT)的偏移量,正负都有可能;第二个是时区的夏令时(DST)规则集,夏令时的规则集指示某个时区是否参与DST,如果是的话,又指示DST何时开始和结束。

使用java.util.Calendar和java.util.TimeZone可以方便地处理与时区相关的日期转换以及夏令时内容。通过TimeZone.getTimeZone(String id)方法,可以创建一个TimeZone对象。这里的id——时区标识通常以时区所包含的最大城市命名。如“America/Chicago”,“China/Beijing”等。通过TimeZone.getAvailableIDs方法可以得到所有可用的时区标识。

下面这段代码演示了如何将美国芝加哥地区的时间转换成GMT标准时间。

```
DateFormat format = DateFormat.getDateInstance(DateFormat.FULL,
DateFormat.FULL, new Locale("en", "UK"));
format.setTimeZone(TimeZone.getTimeZone("GMT"));
Calendar cal = Calendar.getInstance();
cal.setTimeZone(TimeZone.getTimeZone("America/Chicago"));
cal.clear();
cal.set(Calendar.YEAR, 1985);
cal.set(Calendar.MONTH, Calendar.APRIL);
cal.set(Calendar.DATE, 15);
cal.set(Calendar.HOUR, 8);
System.out.println(format.format(cal.getTime()));
cal.set(Calendar.YEAR, 2005);
System.out.println(format.format(cal.getTime()));
```



执行结果如下所示。

```
Monday, April 15, 1985 2:00:00 PM GMT
Friday, April 15, 2005 1:00:00 PM GMT
```

可以看到,由于夏令时规则的改变,芝加哥地区1985年4月15日早8时相当于GMT时间下午2时;而2005年同样的时间则相当于GMT时间下午1时。

使用TimeZone转化时区的一般流程总结如图19-13所示。

使用Calendar创建一个
源时区的Date对象

创建一个目标时区的
DateFormat并用它来
格式化Date对象

```
Calendar cal = Calendar.getInstance();
cal.setTimeZone(.....)
cal.set(.....)
Date date = cal.getTime();
```

```
DateFormat format =DateFormat.getDateInstance(int
dateStyle, int timeStyle,Locale.....);
format.setTimeZone(.....);
format.format(date);
```

这里为DateFormat设置了两个属性,Locale和TimeZone。它们分别独立负责着字符显示格式和时区。设置某一个Locale并不会使DateFormat自动变成对应的时区



图19-13 如何使用TimeZone对象转换时区

如果需要计算处于不同时区的两个Date对象的时间差，可以使用下面的代码。

```
Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
Calendar chinaCal = Calendar.getInstance(Locale.CHINA);
gmtCal.set(Calendar.YEAR, 2007);
gmtCal.set(Calendar.MONTH, Calendar.JULY);
gmtCal.set(Calendar.DAY_OF_MONTH, 4);
gmtCal.set(Calendar.HOUR_OF_DAY, 9);
gmtCal.set(Calendar.MINUTE, 0);
gmtCal.set(Calendar.SECOND, 0);
gmtCal.set(Calendar.MILLISECOND, 0);
chinaCal.set(Calendar.YEAR, 2007);
chinaCal.set(Calendar.MONTH, Calendar.JULY);
chinaCal.set(Calendar.DAY_OF_MONTH, 4);
chinaCal.set(Calendar.HOUR_OF_DAY, 9);
chinaCal.set(Calendar.MINUTE, 0);
chinaCal.set(Calendar.SECOND, 0);
chinaCal.set(Calendar.MILLISECOND, 0);

System.out.println((chinaCal.getTimeInMillis() - gmtCal
.getTimeInMillis()) / (60 * 60 * 1000));
```

两个日期的毫秒值相减，经过计算得到对应的小时差值



在使用TimeZone对日期进行计算时，JDK在处理过程中内置了对所有夏令时计算规则的支持。当某个国家/地区的夏令时规则发生改变时，Sun会及时推出JDK的补丁版本来修正这些规则。因此使用TimeZone来处理时区是一个好的选择。

19.2.1.2 使用资源束(ResourceBundle)进行文本翻译

资源束(Java.util.ResourceBundle)是JDK提供的用于管理程序中地域相关的文本资源(如界面文本等)的功能类。一个资源束对象包含着一套资源的某个版本。因此它有所代表的资源和资源的版本两个关键属性。代表相同资源的资源束拥有一个相同的名字(共享名)，而它所包含资源的版本则由locale属性标识。可以使用ResourceBundle.getBundle(String name, Locale locale)方法获得一个资源束对象。

利用资源束进行国际化工作时，开发者首先为需要国际化的资源编写对应不同Locale的资源束。在使用资源时，根据用户系统的Locale设定用getBundle方法取得对应的资源束对象，并取出它里面的资源用于显示。另外，还需要提供一个默认的资源束，当开发者没有为某个用户所在地域的Locale编写资源束时，系统会用默认资源束来代替，默认资源束的内容通常为英语。

每个资源束都是由一系列的键值对(资源名称-资源值)组成。共享同一个名字的资源束，其包含的资源名称也是一致的，而资源值则根据地域而各不相同。资源束的名称通常由共享名加上地域名称构成。而默认资源束的名称则与共享名相同，图19-14显示了一组共享名为Message的资源束，它包含了两个具体Locale的资源束和一个默认资源束。



图19-14 资源束示例

由图19-14可以看到,资源束的本质就是将一个字符串的不同语言版本存储在相关的一系列文件中,并根据用户所在地域选择对应的版本显示出来。下面的代码演示了如何从图19-14所示的资源束中读取资源并用于显示。

```
ResourceBundle bundle = ResourceBundle.getBundle("Message", Locale.getDefault());

String welcomeMsg = bundle.getString("Message_1");

System.out.println(welcomeMsg);
```

读取共享名为Message的资源束组,并根据用户的默认地
域设定(JVM在启动的时候会从操作系统读取这个设定)选
择对应的资源束对象

从资源束中读取名为Message_1
的资源值并用来显示

资源束是一个抽象类,它的继承结构如图19-15所示。

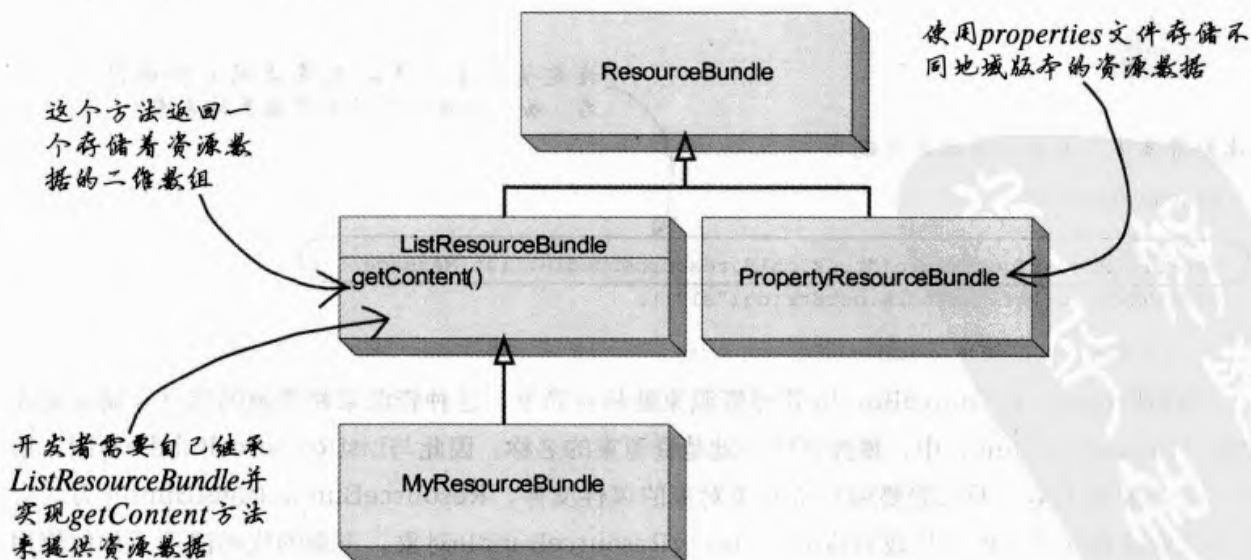
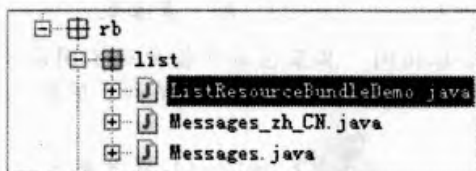


图19-15 资源束的类型结构

图19-15中列出了JDK为ResourceBundle类提供的两个子类，它们分别代表着两种不同的编写资源束的思路。ListResourceBundle采用二维数组的方式组织资源数据，用户需要为每一个资源束生成一个子类，这些子类继承自ListResourceBundle并实现了它的抽象方法String[][] getContents()，所返回的二维数组存储着资源的键值对，有多少个资源束，就需要编写多少个子类。下面的代码分别创建了一个存储着英文信息和中文信息的ListResourceBundle(代码见光盘：\book.ch19.resourcebundle.list.ListResourceBundleDemo)。



```
public class Messages extends ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] {
            { "s1", "OK" },
            { "s2", "Cancel" },
            { "s3", "Retry" }
        };
    }
}
```

每一个资源束都是
ListResourceBundle
的一个子类

```
public class Messages_zh_CN extends ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] {
            { "s1", "确定" },
            { "s2", "取消" },
            { "s3", "重试" }
        };
    }
}
```

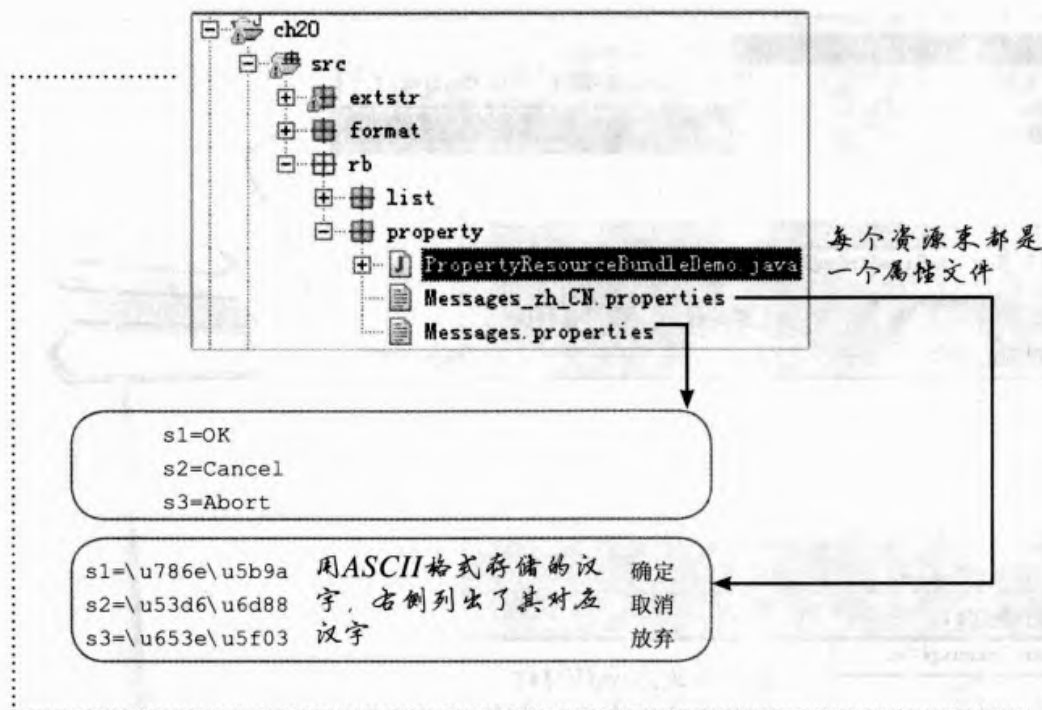
读取资源束，并取得资源的代码

```
ResourceBundle bundle =
```

```
ResourceBundle.getBundle("book.ch19.resourcebundle.list.Messages");
System.out.println(bundle.getString("s1"));
```

指定资源束位置时需要使用完整的包名，否则会抛出找不到资源束的异常

使用PropertyResourceBundle管理资源束就相对简单，这种资源束将资源的信息存储在属性文件（Properties File）中，属性文件名就是资源束的名称。因此与ListResourceBundle不同，用户不需要编写子类，而只需要编写资源束对应的属性文件，ResourceBundle的getBundle方法会自动读取这些属性文件并生成对应的PropertyResourceBundle对象。下面的代码演示了如何使用PropertyResourceBundle实现与上面代码同样的功能(代码见光盘：\book.ch19.resourcebundle.property.PropertyResourceBundleDemo)。



从属性文件中读取资源束的代码与使用ListResourceBundle时完全一致，ResourceBundle将对资源束内容的不同来源的处理逻辑都封装在了其内部。如下所示。

```
ResourceBundle bundle = ResourceBundle.getBundle("book.ch19.  
resourcebundle.  
property.Messages");  
System.out.println(bundle.getString("s1"));
```

※ 注意 : ※

资源束使用ISO-8859-1编码存储和管理属性文件，如果在属性文件中直接存储非ISO-8859-1编码的字符，如中文、日文等，所读取出来的字符串将是乱码，因此在存储这些字符之前，需要将它们转换成Unicode字符的ASCII格式。

Unicode字符的ASCII格式，是为了在旧的文件格式中用ASCII编码存储Unicode字符而设计的，这种格式以“\u”开头，后面跟一个4位的十六进制字符串，该字符串代表该字符在Unicode编码表中的编码，如“你”的ASCII格式就是\u4f60。读取时，ResourceBundle会自动将这个字符串转换成对应的Unicode字符。

JDK提供了一个命令行小工具用来把Unicode字符转换成ASCII格式，native2ascii.exe。可以在{JDK根目录}/bin目录下面找到它。命令格式即为native2ascii SRC_FILE DEST_FILE，也可以在命令行方式下直接运行该程序，输入待转换的文字并按回车键。



为了测试使用资源束的程序是否工作正常，可以通过修改JVM启动时的地域参数来指定JVM所使用的默认地域，图19-16演示了如何在Eclipse中修改这一项内容。

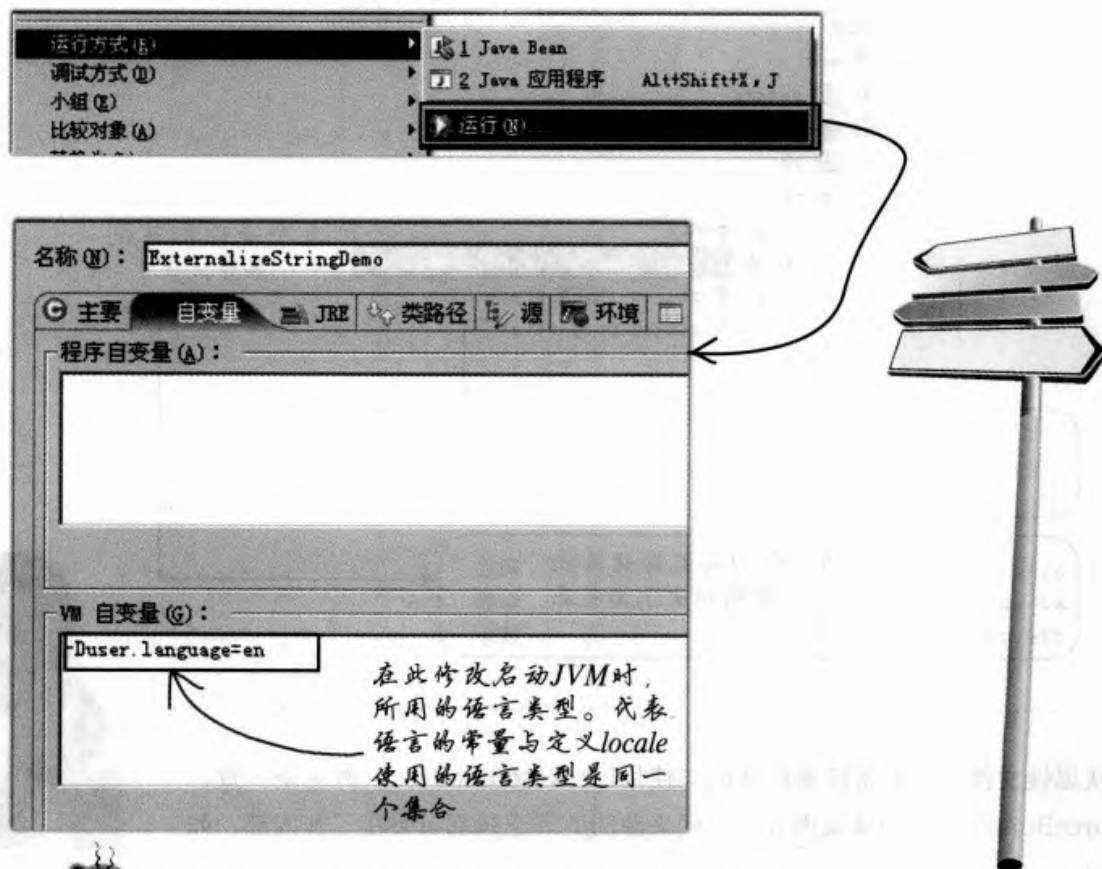


图19-16 修改启动参数来测试外部化字符串的程序

如图19-16所示，在启动JVM时通过设置user.language系统变量可以设置启动时所用的默认Locale，这个变量的取值要求是一个有效的Locale名称，如en,en_US等。

19.2.1.3 Eclipse的外部化字符串功能

对已经开发完成的软件产品进行国际化改造不是一件容易的工作，其中最繁复的部分要算找出程序中硬编码的字符串，并用资源束替代它们了，Eclipse的外部化字符串功能可以帮助开发者进行这部分工作。在编写地址本插件时，很多文本内容，如向导页面中的标题文字等，都是以硬编码方式直接写在程序中的，现在将以创建地址本元素的页面向导为例，演示如何使用Eclipse的外部化字符串功能为这部分内容添加国际化支持。

NewAddressItemWizardPage.java（这是创建地址本元素向导的第一页）的代码如下所示。

```
public NewAddressItemWizardPage() {
    super("新建地址项");
    setTitle("创建地址元素");
    setDescription("创建一个新的地址元素");
    .....
}

public void createControl(Composite parent) {
    .....
}
```

```

final Label label = new Label(container, SWT.NONE);
label.setText("为将要添加的地址元素创建名称。");
final Label label_1 = new Label(container, SWT.NONE);
label_1.setText("设定元素名称:");
final Label label_2 = new Label(container, SWT.NONE);
label_2.setText("为要添加的地址元素指定类别。");
final Label label_3 = new Label(container, SWT.NONE);
label_3.setText("选择元素类别:");

.....
}
public void updatePageComplete() {
    .....

    setErrorMessage("请输入要创建的地址元素的名称");
    setMessage("此名称已存在, 请设置另外的名称!", WARNING);
    setErrorMessage("请选择要创建的地址元素的类别");

    .....
}

```



右键单击要修改的源文件, 选择“源代码”→“外部化字符串”打开外部化字符串功能的对话框, 这个对话框中列出了Eclipse在源文件中探测到的所有字符串, 如图19-17所示。

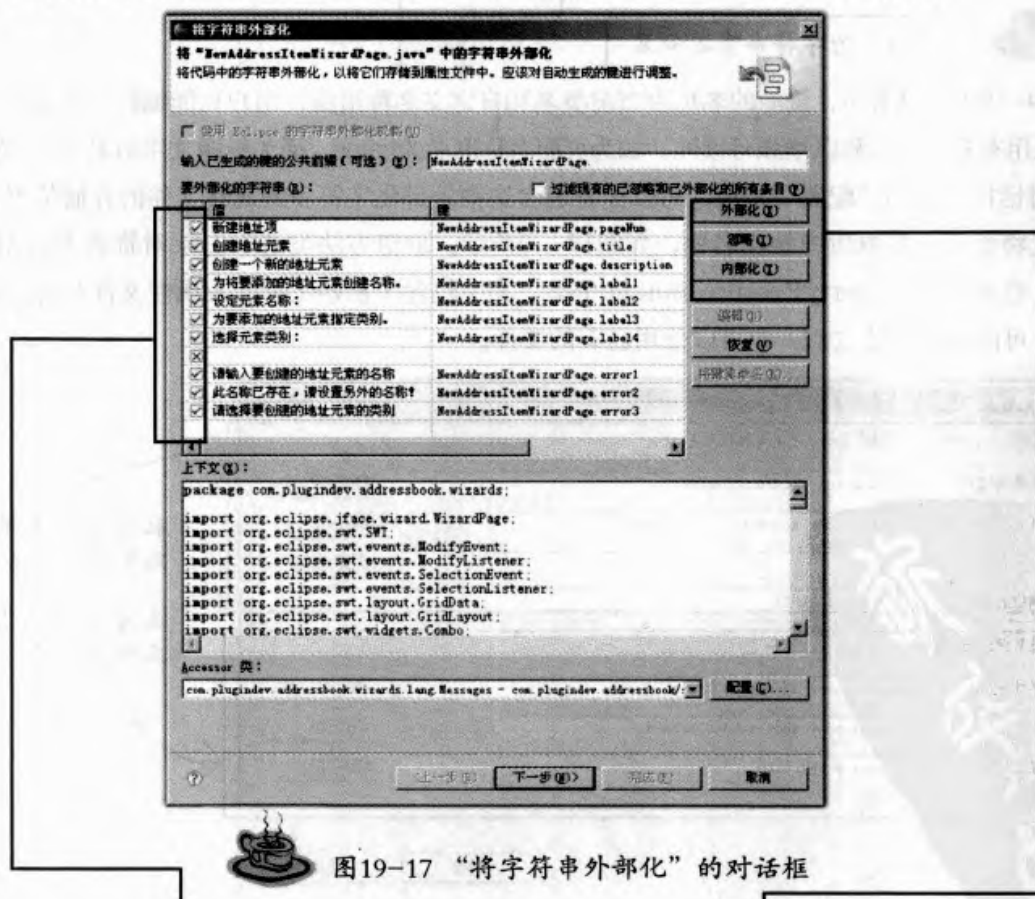


图19-17 “将字符串外部化”的对话框

单击列表中的复选框, 或者选择列表中的字符串然后单击右面的按钮, 可以将字符串标记成以下三种情况之一。

默认的选择 ☒ 是“外部化”，Eclipse会用资源束里的值替代这个字符串，这种情况下，需要为字符串指定一个唯一的标识。

☒ 代表“忽略”或“不要外部化”，对于一些程序内部使用，不涉及显示的字符串，可以选择这个选项。选择该项后，这些字符串将不被更改，而且下次打开这个对话框时，也不会再次显示它们。

☐ 被称为“内部化”或“暂时不要外部化”。如果开发者暂时无法确定是否要外部化某个字符串，可以选择这个选项。下次打开外部化对话框时，它们仍然会被列出来。

为每个字符串选择处理方式后，需要为它们命名，这个名称将作为代表该字符串的资源名称存储在属性文件中，如图19-18所示。

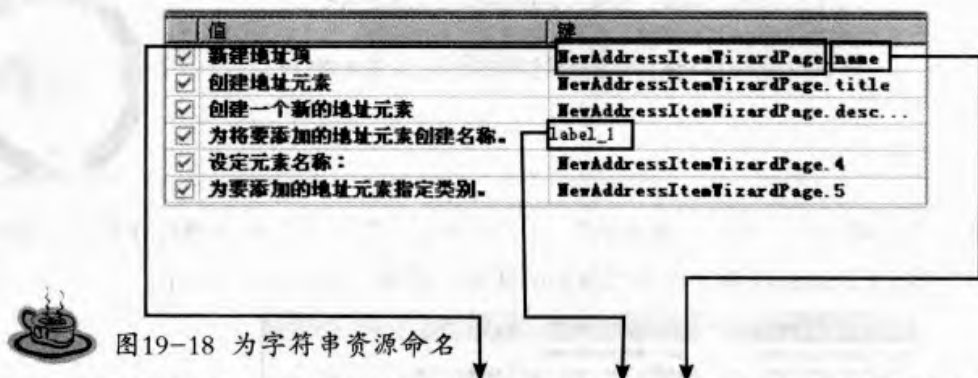
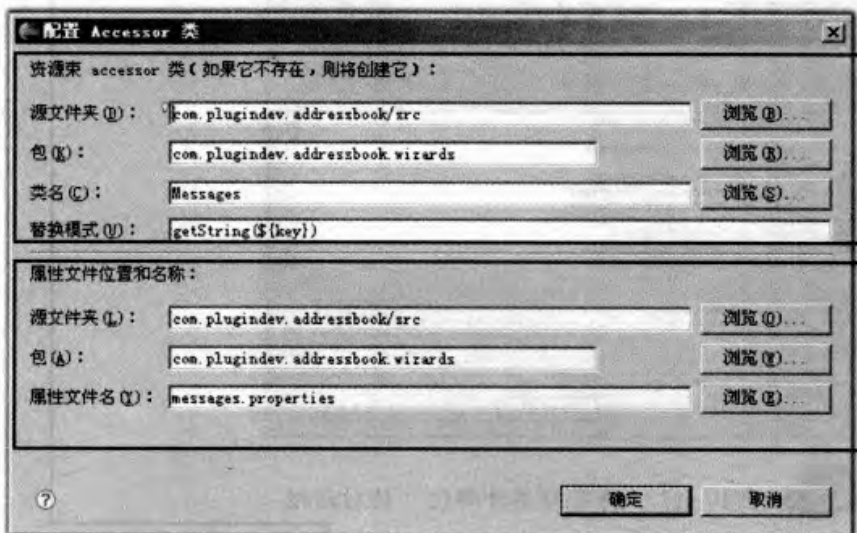


图19-18 为字符串资源命名

在图19-18中可以看到，资源的名称由当前类名和自定义名称组成，用户只能编辑自定义的名称。应该尽量采用有意义的名称以增强可读性，如为页面名称取名为name，而为标题文字取名为title等。

单击对话框下方的“配置”按钮，可以配置包含这些外部化字符串的属性文件的存储位置，同时也可以配置将要生成的取值方法类名称，如图19-19所示。取值方法类是Eclipse对资源束的API做的一个封装，它使用了PropertyResourceBundle子类，其中包含了初始化时读入属性文件和访问属性文件的代码。可以选择新建它们，也可以使用已有的文件。



取值方法类的名称
及所在的包名

属性文件的名称
及所在包名



图19-19 配置属性文件及取值方法类

如图19-19所示，默认情况下Eclipse会将取值方法类命名为“Messages”，而将属性文件命名为“messages.properties”。一般来说，可以采取以下两种方式对这些文件命名。

★如果准备将每个模块（如views模块、editor模块等）的取值类放在不同的路径下面，那么没必要对这些文件特殊命名，直接用默认名称就可以了。

★如果准备将不同模块的取值类和资源属性文件统一放置在同一个路径下面，那么就应该在取值类和资源属性文件的前面加上能够代表模块的前缀。如editors模块的文件可能被命名为editor_messages.properties和EditorMessages等。

在对每一个字符串命名并单击“下一步”按钮预览所要作的改动后，单击“完成”按钮。向导会修改源文件，并生成存储着外部化字符串的属性文件和取值方法类。

被修改后的源文件内容如下（部分）所示。

```
public NewAddressItemWizardPage() {
    super(Messages.getString("NewAddressItemWizardPage.name"));

    //$NON-NLS-1$
    setTitle(Messages.getString("NewAddressItemWizardPage.title")); //$NON-NLS-1$

    setDescription(Messages.getString("NewAddressItemWizardPage.description")); //$NON-NLS-1$
    setImageDescriptor(ImageKeys.
        getImageDescriptor(ImageKeys.IMG_WIZARD_NEW));
}
.....
```

代码中标记说明如下：



原本硬编码在代码中的字符串被改写成了通过取值方法类读取资源的形式



这部分注释内容由外部化字符串的功能添加，用于标识该行中出现的字符串不需要外部化或已经被外部化。

自动生成的取值方法类内容如下所示，它主要包装了创建资源束和读取资源束中字符串的功能，由代码可以看到，它的内容与第19.2.1.2节中介绍PropertyResourceBundle时所写的实例基本相同。

```
public class Messages {
    private static final String BUNDLE_NAME
        ="com.plugindev.addressbook.wizards.lang.messages"; //$NON-NLS-1$
    private static final ResourceBundle RESOURCE_BUNDLE
        = ResourceBundle.getBundle(BUNDLE_NAME);
    private Messages() {
    }
    public static String getString(String key) {
        try {
            return RESOURCE_BUNDLE.getString(key);
        } catch (MissingResourceException e) {
            return '!' + key + '!';
        }
    }
}
```

从资源属性文件中建立资源束

从资源束中读取外部化的字符串

除此之外，工具还自动生成了存储有字符串内容的messages.properties文件，内容如下所示。

```
NewAddressItemWizardPage.name=\u65B0\u5EFA\u5730\u5740\u9879
NewAddressItemWizardPage.title=\u521B\u5EFA\u5730\u5740\u5143\u7684\u7684\u7684
.....
```

下面新建一个属性文件，如下所示，命名为messages_en_US.properties。这个文件将作为英文资源束的内容文件。

```
NewAddressItemWizardPage.name=Create Address Item
NewAddressItemWizardPage.title=Create new address item
.....
```

现在编码的任务已经完成，可以启动程序来看一下效果，为Eclipse程序配置环境变量的步骤与普通Java程序又略有不同，单击“运行”→“运行...”菜单，打开“运行”对话框，在“Eclipse应用程序”项下面新建一个配置，并在左侧的“参数”选项卡中“VM参数”一项中输入“-Duser.language=en_US”以使用美国英语的Locale启动Eclipse程序，现在新建地址项的向导页面如图19-20所示。

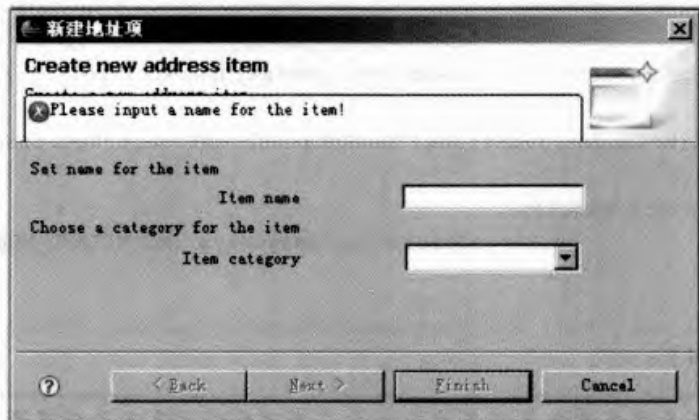


图19-20 英文版的地址向导页面

由图19-20可见，除窗口标题外（该字符串定义在NewAddressItemWizard中，故未被国际化），其他页面文字都已变成了英文，使用类似的方法，读者可以尝试自行国际化其他页面文字。

然而，这种实现方法有个缺陷，包含本地语言信息的资源文件是同程序绑定在同一个插件中，一起打包发布的。因此，如果在插件发布后需要添加对新的Locale的支持，虽然程序代码本身没有做过修改，但却必须重新打包发布插件。使用段（Fragment）可以方便地解决这个问题，在第19.2.2节中，将介绍如何使用段弥补这个缺陷。

19.2.1.4 对Unicode字符的支持

在很多程序员编写的代码中，可以看到如下所示的判断一个字符是否是字母的代码片断。

```
char ch = .....;
if ((ch <= 'A' && ch >= 'Z') || (ch <= 'a' && ch >= 'z'))
    .....
```

这段代码基于字符的ASCII码值来判断它是否是一个字母，因此它只在英文等少数几种基于ASCII字符的编码环境下面可以正常工作；而遇到非ASCII字母，如希腊字母或斯拉夫字母时，程序

就会出错。

这种代码不满足国际化的要求。判断字符类别时，需要使用java.lang.Character中的方法对字符进行判断，如下面代码所示。

```
char ch = .....;
if(Character.isLetter(ch))
    .....
```

Character类中的方法基于字符的Unicode编码对字符进行分组。它考虑到了所有主流语言的情况，因此无论一个字符是日语、中文或是希腊语的字母，isLetter方法都会返回true，如下面代码所示。

```
Character.isLetter('中');
Character.isLetter('に');
Character.isLetter('β');
```

Character提供了许多方法用于判断字符的类型，这些方法都基于Unicode编码。表19-3列出了一些较常用的方法及说明。

表19-3 Character类中的常用方法

方法名	说明
isLetter	判断字符是否是字母
isUpperCase	判断字符是否是大写字母(主要用于拉丁语系，对中文、日文等东亚文字始终返回false)
isLowerCase	判断字符是否是小写字母(同上)
isDigit	判断字符是否是数字
isLetterOrDigit	判断字符是否是数字或字符
isSpaceChar	判断字符是否是空白字符(空格、制表符等)

Character.getType(char ch)方法会直接返回一个字符的类型常量值。用返回值与Character类中声明的类型常量作比较也可以判断一个字符的类型。代码如下所示。

```
if (Character.getType('a') == Character.LOWERCASE_LETTER)
    .....
if (Character.getType('R') == Character.UPPERCASE_LETTER)
    .....
if (Character.getType('>') == Character.MATH_SYMBOL)
    .....
if (Character.getType('_') == Character.CONNECTOR_PUNCTUATION)
    .....
```



19.2.2 国际化支持和段 (Fragment)

本节中，将介绍如何使用段为插件实现国际化支持。

19.2.2.1 什么是段

段是Eclipse插件体系结构的一部分，它依赖于某个插件、作为它的可选组成部分存在，段可以和它所对应的插件分开打包发布，当Eclipse平台装入某一个插件(原始插件)时，会在系统中查找依赖于该插件的段，并将它们一起装入。

在段中，开发者也可以声明扩展点，创建扩展等，这些操作与在插件中进行开发时一般无二，这些内容记载在段项目下的fragment.xml中，在段中同样可以编写Java代码等，图19-21显示了段和它所依赖的插件的关系。

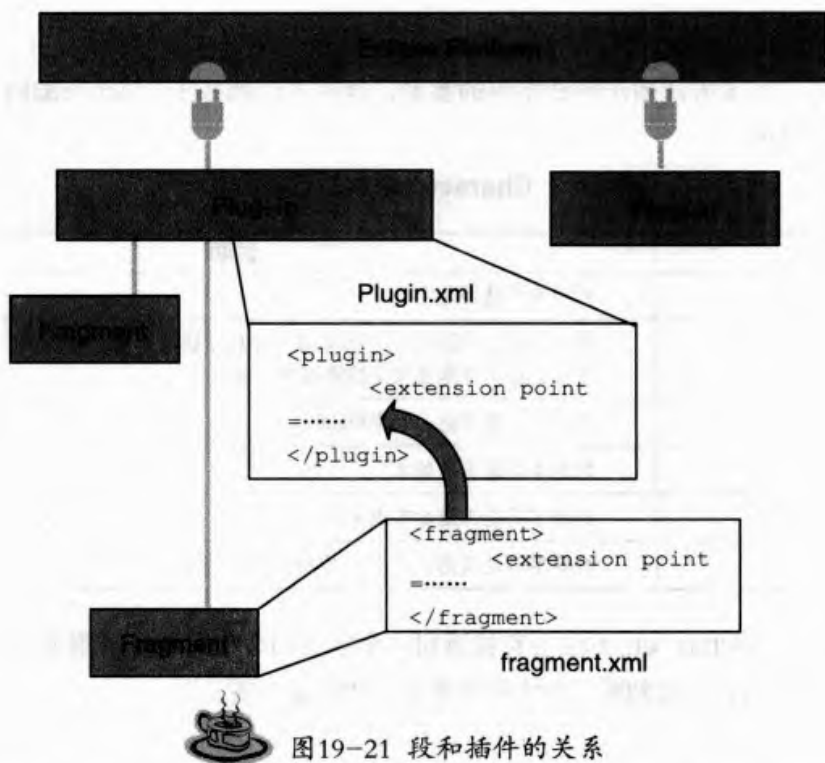


图19-21 段和插件的关系

如图19-21所示，平台装载段时，会把段中声明的这些内容“合并”到原始插件的plugin.xml中，对外部的扩展点使用者和扩展提供者来说，这部分内容是和直接定义在原始插件中没有区别的。原始插件中的Java代码和段中的代码也会被装载到同一个类空间中，如果把段中的包结构设计成和原始插件一样，它们的类就可以互相访问只在包中可见的类和方法，段和插件的关系比插件和插件之间通过扩展点相互交流的关系要更为紧密。

段的依赖性继承自原始插件。也就是说，开发者只要在原始插件中声明了对某个插件的依赖，就不需要在段中重复添加它们，如果段另外还依赖于原始插件所不需要的其他插件，可以另行添加。

段可以用来对已经存在的插件进行扩展，由于它不对原始插件的内容作任何修改，只是单纯地添加一些内容，这种特性使得段特别适合用来管理国际化产品中的多语言数据。开发者可以首先发布一个基本的、只包含少数几种常用语言的软件版本，然后再陆续以段的形式发布新的语言包。用户可以根据需要选择语言包下载，随后只要将这些段复制到自己的程序中，不需要其他任何改动就可以使用新的语言了。在汉化Eclipse平台时，所下载的语言包就是一个段的集合。其中包含了针对Platform, JDT, Help等许多插件的段。

19.2.2.2 使用段实现国际化支持

本节将演示如何利用段为地址本插件实现国际化支持。

在第19.2.1.3节中，已经为地址本插件的向导页面完成了国际化工作，现在开始将英文语言从主插件分离到一个段里。在“新建”对话框中选择“插件项目”→“段项目”来新建一个段项目，段的名字是可以随意取的，并不强制要与其所属的插件的名称有什么关系，但一般出于方便辨认的关系，都会以所属插件的名称加上一个后缀作为其名称。下面所要创建的段为插件提供了多语言支持，因此取名称为“com.plugindev.addressbook.nl”。

创建段的步骤与创建插件相比，多了一步“选择主插件”，如图19-22所示。这里还可以指定需要应用段的主插件最小版本和最大版本，用户下载了段安装到自己的Eclipse环境中后，如果目标环境的主插件版本不在所指定的范围内，Eclipse平台就不会载入这个段，这是为了保证版本兼容性而设计的。

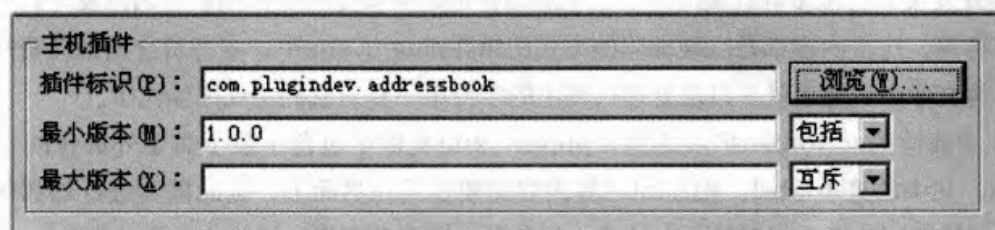
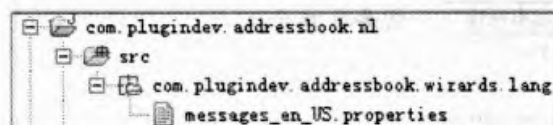
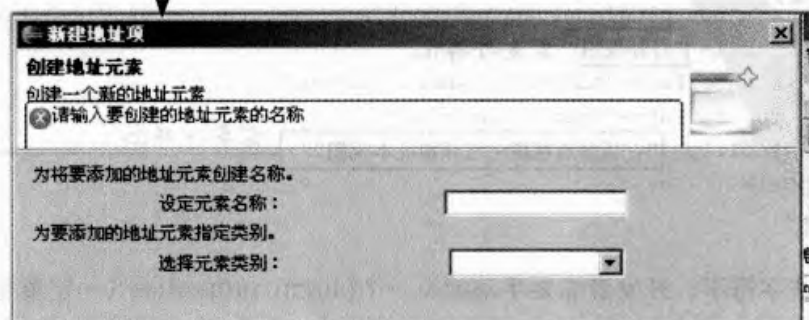


图19-22 创建段时选择主插件

创建了段后，建立与主插件项目相同结构的包层次，然后将中文的资源文件移动到段中。完成后段项目的内容如下所示。



现在可以对段项目进行测试，如图19-23所示，首先关闭段项目，将系统变量改为英文环境，启动地址本插件，这时由于段项目不可用，系统找不到英文对应的资源文件，就会使用默认的资源（中文），在向导页面显示的是中文，而打开段项目后，再次运行插件，向导页面就可以显示英文了。



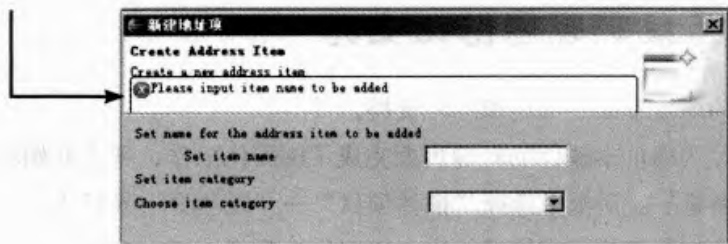


图19-23 支持多语言的向导页面

这样就完成了段的创建，现在原始插件和段语言包可以单独发布出去。

19.2.3 外部化plugin.xml中的字符串

在插件开发中，有很多标签类的内容是在编写扩展时直接提供的，如视图名以及在行为集合中定义的菜单名称等。这些内容都是以硬编码的方式记录在plugin.xml中，本节将介绍如何将这字符串外部化到资源文件中，随后就可以像处理代码中的字符串一样对它们实现国际化了。

以地址本插件的actionSet扩展为例，plugin.xml文件中包含了数个需要外部化的字符串，代码如下所示，其中的菜单label、操作label等内容需要显示在界面上，因此需要进行外部化，而诸如actionSet的id等值因为不需要在界面上显示，只是供开发者参考用，因此不需要外部化。

```
.....
<extension
    point="org.eclipse.ui.actionSets">
    <actionSet
        id="com.plugindev.addressbook.actionSet"
        label="地址本操作" 不需要外部化
        visible="true">
        <menu
            id="AddressBookMenu"
            label="地址本" 需要外部化
            <separator name="content"/>
        </menu>
        <action
            class="com.plugindev.addressbook.actions.OpenAddressViewAction"
            definitionId="com.plugindev.addressbook.commands.openAddressView"
            icon="icons/sample.gif"
            id="com.plugindev.addressbook.actions.OpenAddressViewAction"
            label="打开视图" 需要外部化
            menubarPath="AddressBookMenu/content"
            toolbarPath="Normal/addition"
            tooltip="在当前透视图中打开地址本视图" 需要外部化
        </action>
    </actionSet>
</extension>
.....
```



为了外部化这些字符串，开发者需要手动编写一个plugin.properties（一定要是这个名称，不能更改）文件，放在plugin.xml同样的目录，也就是插件的根目录下面，并在其中包含所有需要外部化的字符串。为plugin.xml中需要外部化的字符串取一个容易分辨的名字作为属性文件的键，通常可以

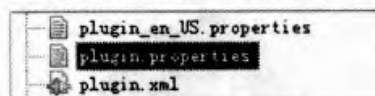
用扩展点的id加上元素id的一部分作为这个键，文件内容如下所示，与前面的例子一样，其中的中文也是用Unicode字符的ASCII编码存储的。

```
actionSet.addressBookMenu.label=\u5730\u5740\u672C
actionSet.OpenAddressViewAction.label=\u6253\u5F00\u89C6\u56FE
actionSet.OpenAddressViewAction.tooptip=\u5728\u5F53\u524D\u900F\u89C6\u56FE\u4E2D\u6253\u5F00\u5730\u5740\u672C\u89C6\u56FE
```

再编写一个资源文件，内容如下所示，命名为plugin_en_US.properties作为英文的资源文件，将这个文件同样放在插件根目录下。

```
actionSet.addressBookMenu.label=Address Book
actionSet.OpenAddressViewAction.label=Open AddressBook View
actionSet.OpenAddressViewAction.tooptip=Open Address Book View in current perspective
```

现在插件的根目录下有了这样两个资源文件，如下图所示。



在plugin.xml中，将需要外部化的字符串替换成“%键名”的形式，Eclipse在读取plugin.xml时会自动用资源文件的内容替换这些字符串，修改后的plugin.xml内容如下所示。

```
<actionSet
    id="com.plugindev.addressbook.actionSet"
    label="%actionSet.addressBookMenu.label"
    visible="true">
    <menu
        id="AddressBookMenu"
        label="%actionSet.addressBookMenu.label">
        <separator name="content"/>
    </menu>
    <action
        class="com.plugindev.addressbook.actions.OpenAddressViewAction"
        definitionId="com.plugindev.addressbook.commands.openAddressView"
        icon="icons/sample.gif"
        id="com.plugindev.addressbook.actions.OpenAddressViewAction"
        label="%actionSet.OpenAddressViewAction.label"
        menubarPath="AddressBookMenu/content"
        toolbarPath="Normal/addition"
        tooltip="%actionSet.OpenAddressViewAction.tooptip" />
</actionSet>
```

被外部化的字符串均以%开始

最后在“构建”选项卡中，将plugin.properties和plugin_en_US.properties文件加入到构建路径中（否则打包发布出去的插件将不会包含这两个文件），这样就完成了plugin.xml中字符串的外部化。现在开发者可以像组织其他资源文件一样，将存储有多种语言的资源文件进行组织并发布出去。如图19-24所示是执行效果图。



图19-24 地址本插件在中、英文环境下的执行效果

19.3 使用功能部件 (feature)

通常，一个基于插件的软件产品出于灵活性考虑，都不会只由一个插件构成，而是设计成相对独立又相互关联的多个插件组合，然而，如果将产品作为分散的插件发布出去，即体现不出其整体性，在版本管理和升级上也有诸多不便，为了解决这些问题，Eclipse提出了功能部件的概念。一个功能部件是由一个或多个插件及它们的附加数据构成的软件产品，如图19-25所示，Eclipse通过功能部件来管理插件的安装、升级等内容。本节将讲述如何开发功能部件，并用它来管理已有的插件。

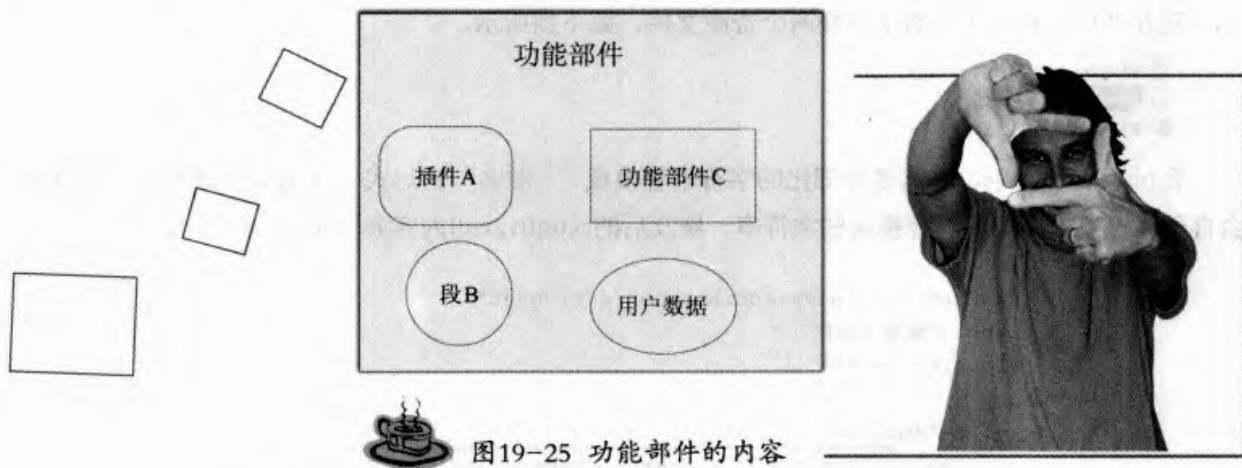


图19-25 功能部件的内容

19.3.1 功能部件概述

使用功能部件可以实现以下这些功能。

★声明使用软件产品的前提需求(需要预安装的其他插件或功能部件)。

★用户可以禁用整个软件产品(禁用后，相应的插件不会被启动)。

★在用户安装了软件产品后，显示欢迎画面。

★在Eclipse平台中显示代表软件产品的标签。

★当产品有了更新版本时，通过自动升级功能下载更新并升级。

一个功能部件主要由以下几部分构成。

★描述 包括对插件的描述，版权声明和许可协议等。

★内容 一个功能插件可以包含插件和段，也可以包含其他的功能部件。

★依赖性 这部分内容描述了该功能部件所依赖的其他插件和功能部件。

★安装 可以指定安装功能部件时的特殊需求(功能部件是否要和别的功能插件安装在同一个目录下，安装功能部件时调用其他处理程序等)。

在Eclipse平台的“帮助”→“软件更新”→“管理配置”菜单中，列出了当前安装在系统中的所有功能部件以及它们之间的依赖关系。用户可以在这里启用、禁用或卸载某一个功能部件，如图

19-26所示。



图19-26 察看系统中已有的功能部件

开发者可以为功能部件设定标签，当用户安装了功能部件后，它的标签会显示在“关于Eclipse SDK”对话框中。单击这个标签会列出功能部件的详细信息，如图19-27所示。这样对于系统中安装有哪些功能部件，用户就比较一目了然了。在第19.3.4节中将介绍如何为插件设定标签。



图19-27 在“关于Eclipse SDK”对话框中添加功能部件的标签

创建了功能部件后，下一个问题就是如何将它发布给用户了。旧有的做法是将功能部件打包发布，用户自行解包到本地的Eclipse环境中；在现在的Eclipse环境中，这种方法仍然可行，但是一种推荐的做法是使用更新站点来安装和升级功能部件，与前者相比，使用更新站点有如下优点。

★用户能够方便地添加和卸载、启用和禁用功能部件，而不需要手动添加或删除文件。

★安装时，平台会自动检测功能部件的依赖。在依赖不满足的情况下，会拒绝安装这个功能部件。这使得用户对于功能部件的可用性有更清晰的了解。

★Eclipse的自动升级功能需要更新站点的支持，建立一个更新站点，用户就可以在新版本发布时方便地升级到功能部件的最新版本。

在第19.3.3节中，对建立与使用更新站点的内容有详细的叙述。

19.3.2 创建功能部件

本节将创建一个包含了地址本插件及它的段的功能部件。

功能部件不包含可执行代码，它的开发主要是一个配置的过程。选择“文件”→“新建”→“项目”菜单，打开“新建项目”对话框，在“插件项目”类别下面选“功能部件项目”，为功能项目

输入名称“com.plugindev.address”后，单击“下一步”按钮，在插件列表中选择地址本插件和它的段，单击“完成”按钮，Eclipse会自动打开feature.xml这个文件，其中包含了功能部件的各种信息。

在“概述”选项卡中，主要包含着功能部件的标识、版本和名称等一般信息。Eclipse的版本号由三位数字构成，分别表示主版本号，小版本号和微版本号，之间用点分隔，如1.0.3是一个有效版本号。这里还可以指定更新站点的URL和名称，如果希望为功能部件添加自动更新的功能，会使用到更新站点，在第19.3.4.3节“更新站点与自动更新功能”中，对此有详细的讲述。

“信息”选项卡包含了功能部件的描述、版权声明、许可协议以及更新的备选站点等信息。在安装功能部件时，会显示许可协议要求用户确认，在系统中通过管理配置察看和管理功能部件的时候，也会显示这些内容。发布开源软件时，可以使用Eclipse社区为开源软件提供的Eclipse公共协议(Eclipse Public License,EPL)，以下网址包含EPL协议的全文。

<http://www.eclipse.org/legal/epl-v10.html>

在“要访问的站点”中，可以为“概述”选项卡中指定的更新站点添加备选站点。当升级时如果访问主站点失败，也会尝试访问这些备选站点。

“插件”和“包括的功能部件”选项卡分别用来管理功能部件中所包含的插件和子功能部件，它们是开发功能部件过程中最主要的部分。

这里需要对列表中插件和功能部件的版本号稍微做一下说明，如果读者比较细心，会发现无论所选择的插件当前版本是多少，插件清单中显示的插件版本号都是0.0.0，如图19-28所示，这是由于插件的同步选项选成了“构建时使版本同步(建议)”，这时插件清单中不会显示插件的真实版本号，而全部以0.0.0代替。当开发者导出功能部件时，Eclipse会取得插件的最新版本号并打包。

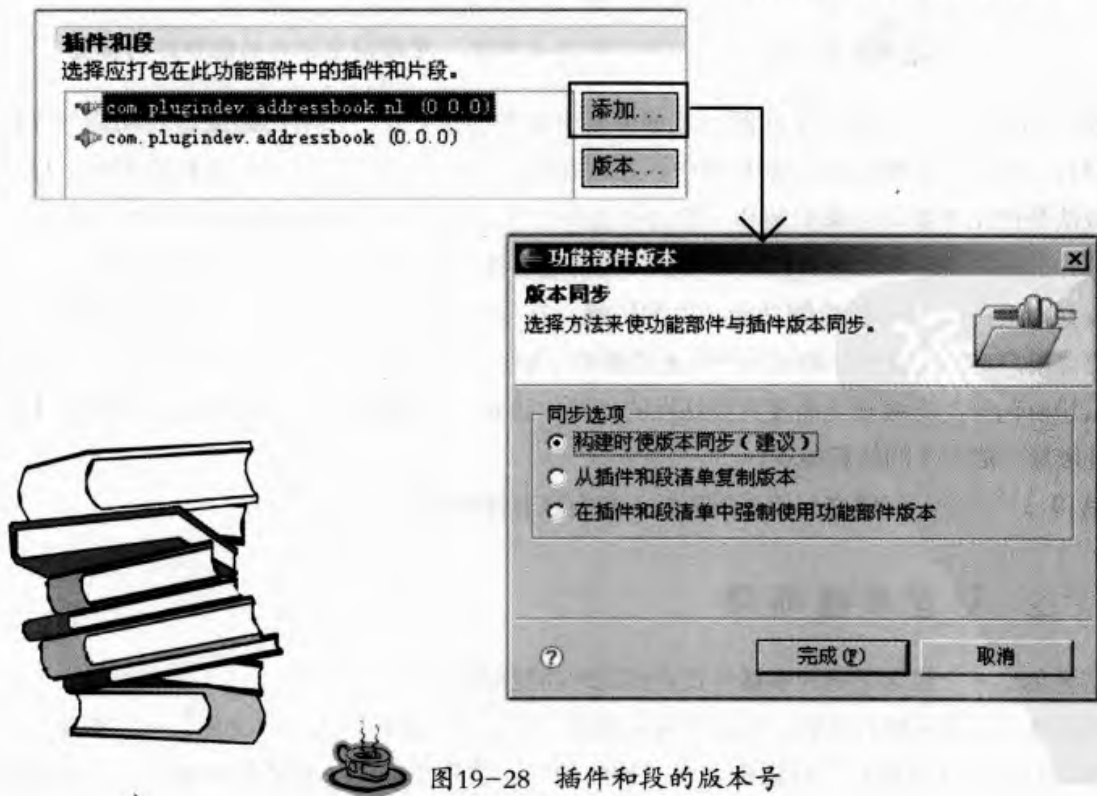


图19-28 插件和段的版本号

如图19-28所示,关于版本还有另外两个选项,“从插件和段清单复制版本”和“在插件和段清单中强制使用功能部件版本”。选择前者会将插件/段的真实版本号显示在插件清单中,但是如果在创建功能部件之后插件的版本又有更改,功能部件的清单不会随之更新。这样导出功能部件时,会导致找不到指定的插件。如果选择后者,则功能部件会将自己的版本号赋到所有它包含的插件上去,这会修改插件的plugin.xml。当开发者向功能部件中添加一个插件而增大功能部件的版本号时,所有原来未改动的插件版本号也会随之上升。这两个选项有可能导致版本号管理的混乱,必须谨慎应用。

如果所创建的功能部件依赖于其他插件或功能部件(计算逻辑是功能部件所包含的插件依赖于某个插件或功能部件),可以在“依赖性”选项卡中指定它们。指定了依赖关系后,在通过更新站点安装该功能部件时,如果目标系统上没有依赖性中指定的插件,安装会无法继续进行。这可以防止用户安装尚不能使用的功能部件。单击“计算”按钮,清单中会自动按照这个计算逻辑列出所有依赖的插件。开发者也可以手动添加其他插件,但是一般没有必要。

※ 注意: ※

依赖性的检查仅仅对通过更新站点安装的功能部件有效,如果用户手动将功能部件的文件复制到本地Eclipse中运行,这个检查是不会生效的,这种情况下,运行插件时会出错。开发者应该考虑到这种情况,对用户给出相应的错误提示。



在“安装”选项卡中,可以设定安装时的各种选项。如指定功能部件对安装目录的要求,为功能部件添加非插件数据等。如果功能部件中包含有不属于任何一个插件的数据,如不同语言版本的版权信息等,可以在这里将它们添加到功能部件中。“安装处理程序”可以用来执行非标准的安装任务,更新管理器一旦下载了安装处理程序,这些处理程序就开始处理数据条目。另外还可以指定功能部件对安装平台的要求。

将这些内容都配置好后,功能部件的创建就完成了。

19.3.3 更新站点与自动更新功能

Eclipse通过更新站点来检测和安装可用的功能部件及更新。本节将创建一个更新站点,并通过它将上一节中所创建的功能部件发布出去。

19.3.3.1 创建更新站点

更新站点项目同样不包含可执行内容,创建它也主要是一个配置的过程,在“新建项目”对话框中,选择“更新站点项目”以创建一个更新站点项目,此处将更新站点命名为“com.plugindev.address.update”,更新站点项目的核心是site.xml文件,用站点清单编辑器打开它,在“站点图”选项卡中,可以创建一些类别,然后将功能部件添加到这些类别下面,也可以直接将功能部件添加到站点中,在从该更新站点安装功能部件时,未分类的功能部件其类别将显示为“其他”。

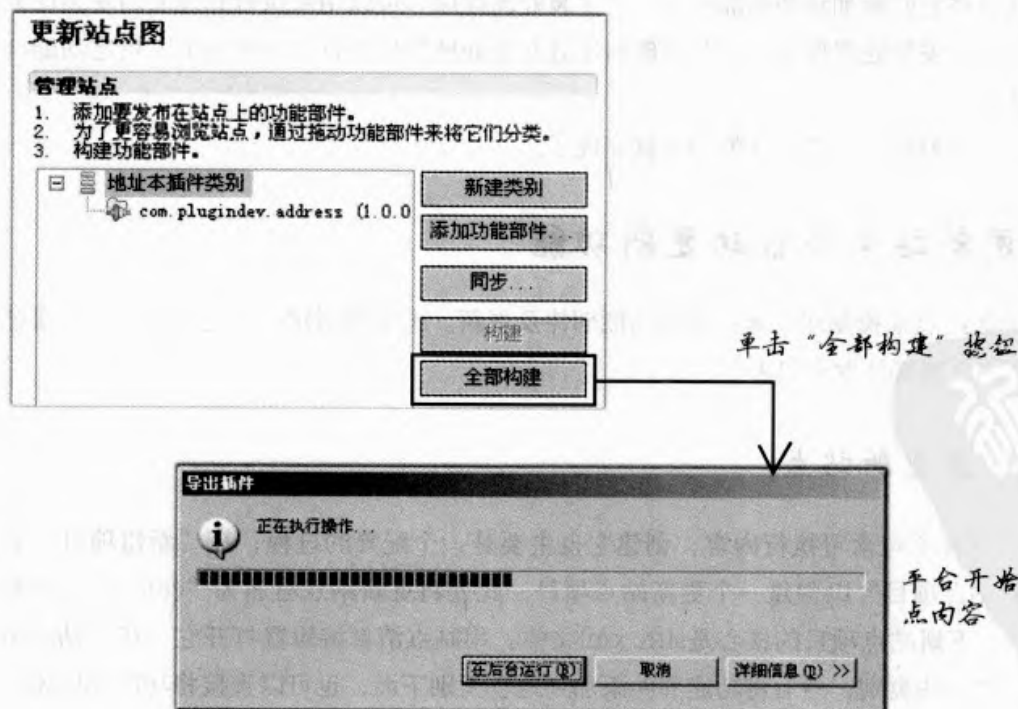
在下一个选项卡“归档”中,可以设定更新站点的URL和描述信息。URL可以是本地路径,也可以是通过HTTP等协议访问的网络路径,此处设定成本地文件目录D:/updatesite,这个地址需要和

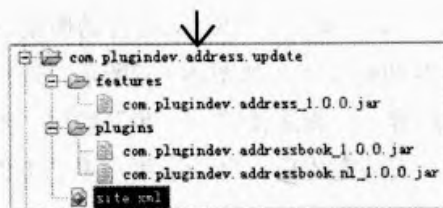
功能部件的更新站点地址相同,因此同时也要修改功能部件的更新站点,如图19-29所示。



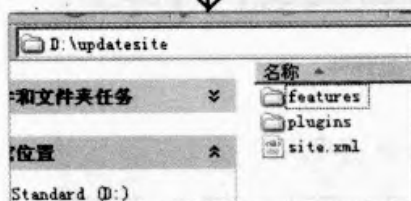
图19-29 设置功能部件的更新站点

设定好这一切后,可以开始构建更新站点了。在“站点图”选项卡中选中某一个功能部件单击“构建”按钮,或者直接单击“全部构建”按钮,Eclipse会在当前目录下生成feature和plugins目录,并将所有相关的功能部件和插件导出到其中,整个流程如图19-30所示。





构建完毕后，在更新站点项目的目录下生成了对应的插件和功能部件文件



手动将features和plugins的文件夹以及site.xml文件复制到更新站点对应的位置



图19-30 为更新站点添加功能部件

将所生成的两个目录连同site.xml复制到指定的更新站点URL，就完成了更新站点的创建。

19.3.3.2 安装功能部件与查找更新

下面演示如何通过刚才创建的更新站点安装功能部件。单击“帮助”→“软件更新”→“查找并安装”菜单，打开Eclipse的升级管理器，然后选择选择“搜索要安装的新功能部件”，单击“下一步”按钮，在打开的对话框中，会列出所有已配置的更新站点，在这里需要将自己的更新站点加入进去，如果是通过网络访问的站点，单击“新建远程站点”并指定站点的URL，如果是本地站点，则单击“新建本地站点”并指定包含site.xml文件的文件夹，如图19-31所示。

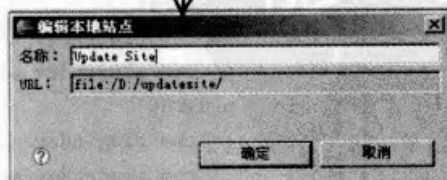
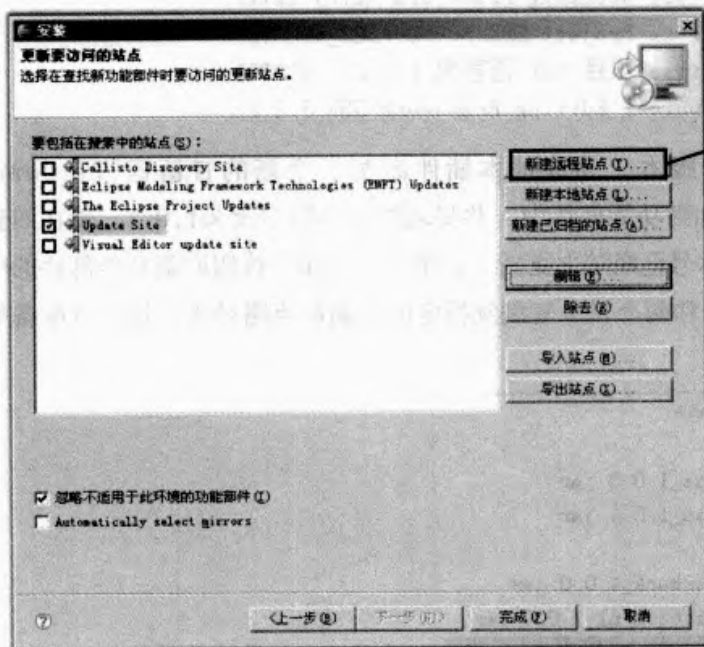


图19-31 通过本地站点安装功能部件

在列表中选择需要搜索的站点后，单击“下一步”按钮，Eclipse会自动搜索站点并列出所有找到的功能部件，这里应该找到刚才所创建的地址本功能部件，如图19-32所示，选择它并单击“下一步”按钮，Eclipse会列出功能部件的版权声明等内容，一直选择“下一步”按钮，然后单击“完成”按钮，在用户确认安装后，安装过程将自动进行。安装完成后，会提示用户重启工作台。重启后安装过程就完成了。

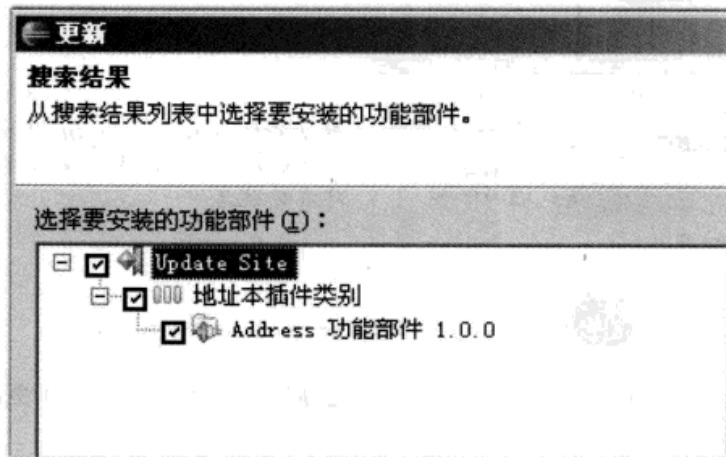
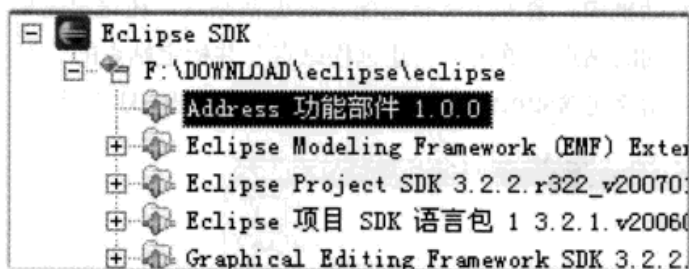
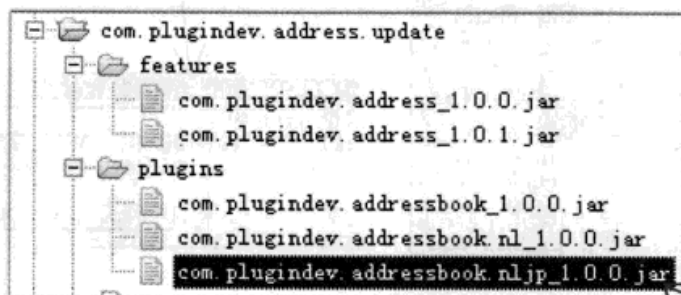


图19-32 在更新列表中选择地址本功能部件

这时打开“关于Eclipse SDK”的对话框或“管理配置”对话框，都可以看到新安装的功能部件，如下所示。



下面演示如何为功能部件创建更新版本。为地址本插件添加一个新的日语语言包com.plugindev.addressbook.nljp，并将它添加到功能部件中，并将功能部件版本改为1.0.1，现在回到更新站点项目中，站点图中功能部件的版本号已经随之更新了，单击“同步”按钮刷新功能部件的内容，并重新构建一下，将新生成的site.xml和两个目录复制到指定的更新站点路径下，这样就准备好了新版本的功能部件。用户可以进行更新了。



新增的段项目

现在单击“帮助”→“软件更新”→“管理配置”菜单，展开列表找到地址本插件的功能部件，

单击右侧的“查找更新”，Eclipse就会列出所找到的1.0.1版的地址本功能部件并提示安装，如图19-33所示。

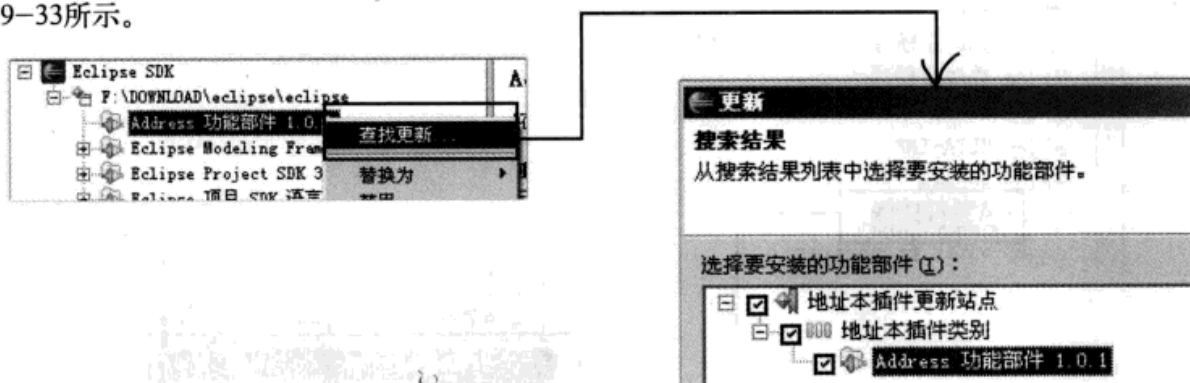


图19-33 为地址本插件查找更新

如果用户在Eclipse中启用了自动更新功能，自动更新运行时也会找到这个新版本的功能部件。

19.3.4 添加产品标签 (branding)

如果为功能部件添加了产品标签，在用户平台的“关于Eclipse SDK”对话框中会为该插件显示一个图标，提醒用户已安装的功能部件产品信息等。为功能部件添加产品标签的过程分为两步，首先需要在feature.xml中指定一个品牌插件(Branding Plug-in)，产品标签的信息将从这个目标插件中取得，这里将品牌插件指定为com.plugindev.addressbook，如图19-34所示。

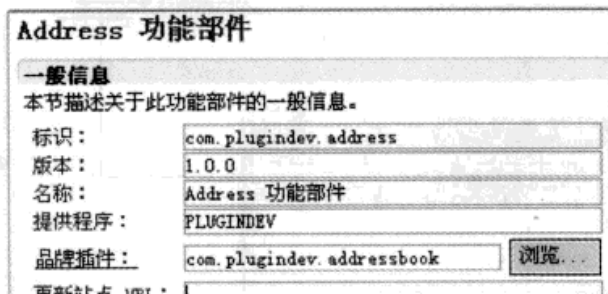


图19-34 为功能部件指定一个品牌插件

在指定的品牌插件中，在根目录下手动添加一个配置文件about.ini，并在其中添加如下两行内容。

```
aboutText= \u5730\u5740\u672c\u529f\u80fd\u90e8\u4ef6
featureImage=image.gif
```

插件启动时，Eclipse会把该文件当做一个属性文件读取(使用java.io.Properties)并相应配置功能部件，因此，在其中使用到中文等Unicode文字时，同样需要使用Unicode的ASCII格式表示，about.ini的文件内容及其作用如图19-35所示。

19.4 动态插件

下面简要讲述一下动态插件的基本概念以及如何开发动态插件。

从Eclipse 3.0开始,首次提出了动态插件的概念。在早期的Eclipse版本中,由于运行时框架的能力限制,框架中用于记录系统中存在着哪些插件的注册表不能动态更新,只能在框架启动时一次性计算出来,因此用户每次安装或卸载插件时都必须重新启动整个工作台。从3.0开始,这一情况得到了改进,基于OSGi的框架Equinox被加入到了传统的运行时框架中,OSGi是完全支持动态更新模块注册表的,因此Equinox的加入,使得动态向系统中添加/删除插件成为可能。

然而,框架的支持并不意味着插件开发者可以透明地享用这个新功能特点,插件必须响应系统在运行时发出的动态通知,并做出适当的反应;另外向系统中动态添加插件或移除插件,受影响的不仅仅是这个插件本身,还有那些依赖于它的插件,进入“动态”时代后,插件们根据它们是否能够意识到这种动态改变并相应地响应这改变,被分为4种类型。

第一种是非动态类型。这是指那些完全不支持动态改变的插件,Eclipse 3.0之前开发的旧插件通常都属于这个类型。用户可以通过新的运行时框架动态添加/删除这些插件,但是框架却无法通知它们这种改变,如一个插件在运行时被移除了,但是它却没有恰当地清理自己使用的资源或取消它在其他插件上注册的监听器等,这些都会导致插件无法被完全移除掉,因此试图动态修改一个非动态类型的插件是很危险的,可能会有预想不到的副作用。非动态的插件也同样无法意识到其他插件的变化,如其他插件在系统启动后动态地在它的某个扩展点上注册了扩展,非动态插件无法意识到这个改变,因此会忽略掉这个动态扩展。

第二种是被动动态类型。被动动态类型的插件可以意识到其他插件的动态改变,如其他插件在自己身上注册了一个新的扩展等;但是它本身可能还不支持动态安装或卸载。比较基础的系统级插件大都属于这种类型,如org.eclipse.ui等,这些插件通常处在一个运行平台的底端,作为基础部分而存在,它们不能被移除,因此也就没有必要被设计成可动态移除的类型。

第三种类型是主动动态类型。主动动态类型的插件允许框架在运行时动态地添加或删除它们,这些插件会监听事件并在启动和关闭时相应地做出资源分配,资源清理等反应,但是这些插件却不一定能对它们所交互的插件的动态变化做出恰当的反应,处在一个运行体系中上层的应用插件都应该至少满足这一类型的要求。

最后一种类型则是全动态类型。这种类型是被动动态类型和主动动态类型的合集,它们的功能最为强大,可以应付任何动态变化的情况。它们或它们所交互的插件都可以被动态地添加或删除,这种类型的插件可以在运行时被动态升级,而不需要重启整个系统,因此从理论上讲,全部由全动态类型的插件构成的系统是可以永久跑下去而不需要停机的,因为无论哪一部分出了故障,都可以动态地将它升级到一个好的版本来替换。

一个插件在运行时是否真的能够做到动态,除了和它本身的实现相关,和它所交互的那些插件也有极大的关系。举例来说,即使一个插件是全动态类型,如果与它交互的插件中有非动态类型的插件,那么它在运行时也是无法完全移除自己的,只要那个非动态类型的插件还持有对这个动态插件中某一部分的引用,这个动态插件就仍然会滞留在内存中。

当动态添加/移除的事件发生时,即使涉及的某些插件是不支持动态变化的,运行时框架也仍然会尝试找到办法来尽量完成这个变化,然而,如果涉及的插件对动态变化没有很好地支持,那么整个

系统会很容易出现错误或不希望发生的副作用，因此在开发插件时，应该尽量全面考虑这些因素，并相应地配置自己插件产品的动态行为。

首先，对于被动动态类型的插件而言，必须做到当其他插件从系统中被删除时，要放弃一切对这个插件的引用，包括在其中定义的类、资源等，如果插件还持有通过扩展点从其他插件创建出来的类型实例，在目标插件将被移除时也要记得销毁这些引用。为了做到这一点，插件需要在系统维护的注册表上注册一个监听器，这样当其他插件被移除时就会得到通知了。

通常为了便于处理扩展，有些插件会在自己内部维护一个扩展的缓存，那么当扩展发生动态变化时，插件也要记得更新这个缓存。下面用一个简单的缓存实例演示如何将地址本插件改造成开发全动态类型的插件，示例在地址本插件中加入一个缓存类ExtCache来保存所有扩展了messages扩展点的插件，这个缓存类使用一个HashSet来存放在自己所属插件的扩展点上注册过的扩展。代码如下所示（代码见光盘：\book.ch19.com.plugindev.addressbook.cache.ExtCache）。

```
public class ExtCache implements IRegistryChangeListener {
    private static final String PID = "com.plugindev.addressbook";
    private static final String PT_ID =
        PID + "." + "messages";

    private final HashSet extensions = new HashSet();
    .....
}
```

代码中关键词说明如下：

缓存类实现了IRegistryChangeListener接口，这个接口可以用来监听插件注册表的变化。

将插件名和扩展点名声明为常量。

在插件启动时，调用缓存的startup来注册所有静态扩展自这个扩展点的扩展。同时将自己注册到系统的插件注册表上，开始监听事件。代码如下所示。

```
public void startup() {
    IExtensionRegistry reg = Platform.getExtensionRegistry();
    IExtensionPoint pt = reg.getExtensionPoint(PT_ID);
    IExtension[] ext = pt.getExtensions();
    for (int i = 0; i < ext.length; i++)
        extensions.add(ext[i]);

    reg.addRegistryChangeListener(this);
}
```

代码中关键词说明如下：

1 取得静态注册的扩展，并将它们加入缓存中。

2 开始监听注册表的变化。

在IRegistryChangeListener接口中，只有一个registryChanged方法需要实现，当注册表发生变化时它将被调用。代码如下所示。

```
public void registryChanged(IRegistryChangeEvent event) {
    IExtensionDelta[] deltas =
        event.getExtensionDeltas(PID, PT_ID);
    for (int i = 0; i < deltas.length; i++) {
        if (deltas[i].getKind() == IExtensionDelta.ADDED)
            extensions.add(deltas[i].getExtension());
        else
            extensions.remove(deltas[i].getExtension());
    }
}
```

代码中标记说明如下:

如果有动态添加的扩展, 向缓存中添加; 如果有动态删除的扩展, 从缓存中删除。

现在这个插件就已经符合了被动动态类型的要求。下面再做最后一步使其符合主动动态类型的要求。为缓存类添加一个shutdown方法, 并在插件的stop方法中调用它。这个方法里清除了所有记录于缓存中的信息, 并且停止监听插件注册表。代码如下所示。

```
public void shutdown() {
    extensions.clear();
    IExtensionRegistry reg = Platform.getExtensionRegistry();
    reg.removeRegistryChangeListener(this);
}
```

通过这个小例子, 读者可以领会到开发一个全动态插件所要做的所有事情。

在开发动态插件时, 除了扩展点外, 还要特别注意静态的容器或字段等地方有没有持有其他插件的引用。如果持有, 在目标插件变化时也要记得更新这些地方。

开发一个动态插件所要做的事情, 和前面讲述SWT资源时提到的“谁分配谁释放”的原则有一点相似, 都是要对自己申请来的“资源”负责管理。一般来说, 在编写插件的shutdown函数时, 需要按照以下的顺序检查资源的释放情况。

★如果正在监听其他插件的状态或系统级的状态变化, 要记得移除这些监听器。一个例外是注册在SWT控件上的监听器不需要手动移除, 因为当窗口关掉时, dispose方法会将这些监听器直接销毁。

★如果持有SWT资源, 释放它们。

★其他系统资源, 如文件句柄, 网络连接等, 关掉或释放它们。


★从其他插件把包含本插件信息的内容移除。比如其他插件在本插件上注册的监听器等。如果有在工作区中存储的一些信息, 也要记得删除。

★如果有其他手动申请或注册的内容, 都要记得手动移除它们。

★如果插件启动了后台进程或工作, 在退出前要停止它们。

19.5 本章小结

本章在第二部分插件开发的基础上，讲述了一些插件开发的高级技巧。这些内容或许在个人开发过程中不甚重要，但是对于企业级的产品开发或想要开发比较正式的软件产品的读者都是非常有用的，读者在掌握了基础的插件开发后，就应该学习并实践一下这些内容。下一章将介绍RCP程序的开发过程，以开发脱离Eclipse平台独立运行的程序。



OK，本章轻松通过了！庆贺一下吧！

拉开崭新的学习帷幕

第20章 富客户端平台 (RCP) 技术

富客户端是目前流行的一种架构体系。它通常指具有独立用户界面的客户端程序。富客户端程序可以是依赖于服务器的前台界面，也可以是独立运行的软件系统，与传统客户端程序相比，富客户端程序的功能更为强大，用户界面体验也更为丰富。Eclipse的富客户端平台(Rich Client Platform, RCP)基于插件技术体系，为开发者提供了许多强大和方便的功能。本章将简要地介绍RCP的体系架构以及如何基于RCP开发富客户端程序。

本章内容包括：

- ★富客户端技术介绍。
- ★RCP平台架构。
- ★开发RCP产品。
- ★将插件改造成RCP程序。



进入第20章

20.1 富客户端技术介绍

通常所说的富客户端程序，主要是指具有以下特点的图形界面程序。

★拥有本地化的窗口界面，并且支持拖放操作、任务栏图标等与图形界面系统紧密相关的功能。它能够为用户提供丰富的图形界面体验。

★平台无关，不需要修改源代码就可以部署并运行在各种主流图形平台上。

★自动升级功能，当程序出现新版本时，可以自动下载并升级本地程序系统。

目前流行的富客户端程序有以下几种架构。

独立型，如图20-1所示。

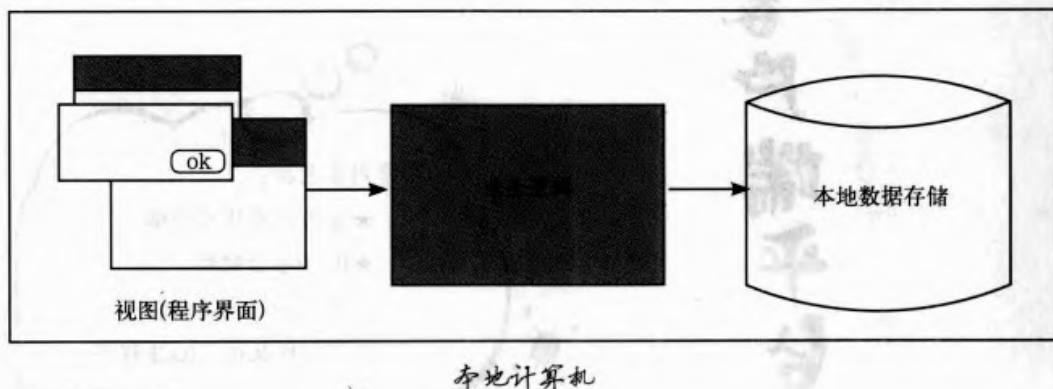


图20-1 独立型富客户端架构

这一类型将所有业务逻辑放在本地程序中，使用文件系统或本地数据库系统存储数据。整个系统与外部世界没有任何交互。许多桌面应用软件，如Office Word, Adobe Photoshop等都属于这一类型。

二层型(客户-数据库型)，如图20-2所示。

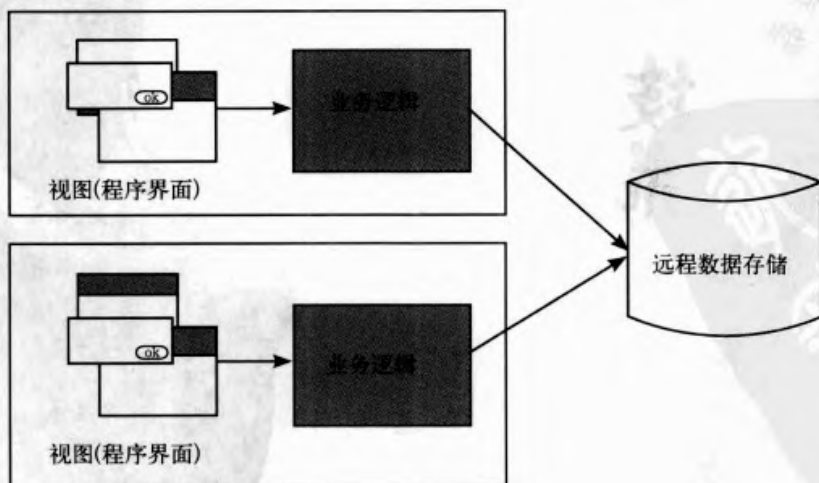


图20-2 二层型富客户端架构

这一类型的架构中，业务逻辑仍然存留在客户端程序中，而数据存储部分则被从各个计算机系统中抽取出来，存放于统一的数据存储系统中（通常是数据库）。各台客户系统通过网络与数据存储系统相连。

这种类型缺乏集中的错误检测机制。如果某一个客户端的程序错误，则会导致整个系统数据混乱，因此目前很少采用。

三层型（客户-服务器型），如图20-3所示。

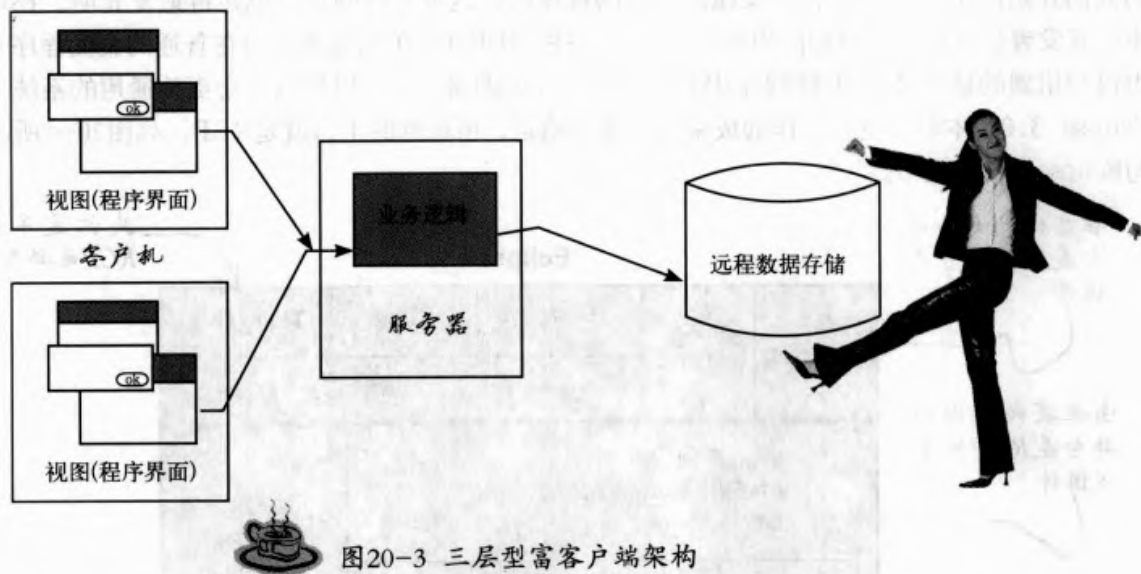


图20-3 三层型富客户端架构

这种架构将业务逻辑部分从客户端剥离出来，转移到服务器上。服务器负责访问数据库系统，并通过Web Service等方式向客户端提供服务，而客户端只负责程序界面的显示和操作。

这种架构将显示逻辑和业务逻辑相剥离，程序设计的层次性更明确，安全性也更高；其缺点在于程序运行完全依赖于网络服务，如果网络瘫痪，程序则无法正常运行。比较适用于有稳定网络连接的环境，如大型企业内部使用。

复合型，如图20-4所示。

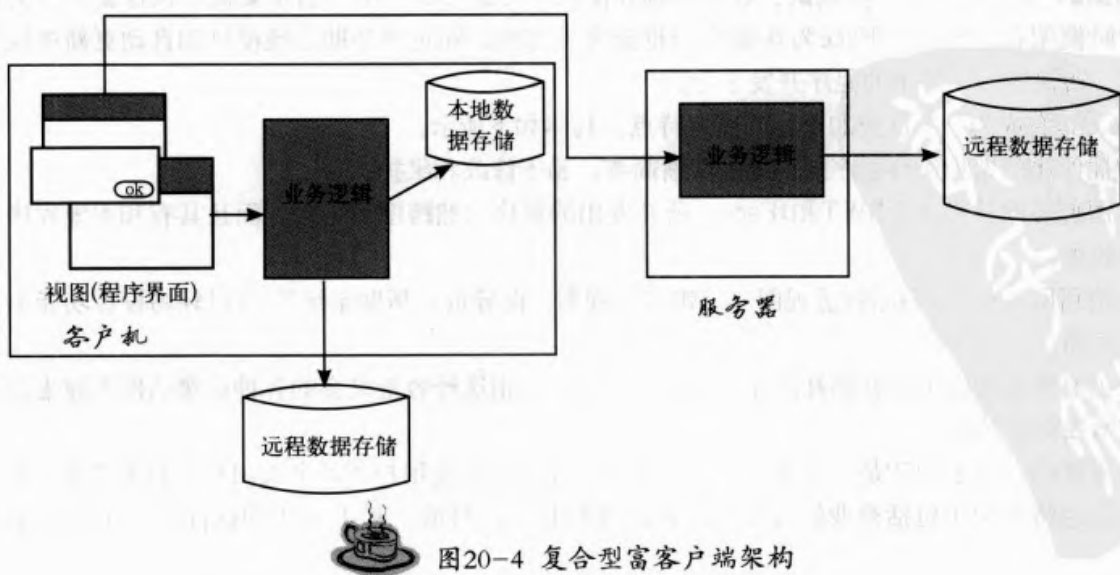


图20-4 复合型富客户端架构

复合型架构吸收了上述几种架构的优势。它拥有本地存储系统,在网络瘫痪时可以正常使用;在网络恢复时,又可以通过服务器与其他客户端进行同步,是目前最流行的富客户端程序架构模式。

可见,无论是哪一种形式的架构,都离不开功能强大的用户界面,而在Eclipse出现之前,Java程序语言所缺乏的正是用户能够接受的图形界面开发平台,Eclipse RCP的出现,弥补了这一空缺。RCP是基于Eclipse平台技术的一个富客户端程序开发平台,开发Eclipse平台的最初目的,是创建一个通用的IDE开发平台,然而逐渐地,有人开始尝试用它开发一些其他用途的插件。当越来越多的人开始使用Eclipse平台开发他们自己的程序时,这项工作的前景也显得愈发光明。在这个过程中,开发者们开始精简平台的内容,他们将一些与IDE开发关联紧密,但在普通的桌面程序开发过程中很少用到的插件(如版本控制等)从IDE开发平台中剥离开来,以得到一个更为通用的系统平台,到Eclipse 3.0版本时,这项工作的成果已经基本稳定,所得到的平台就是RCP,如图20-5所示为RCP与Eclipse平台的关系。

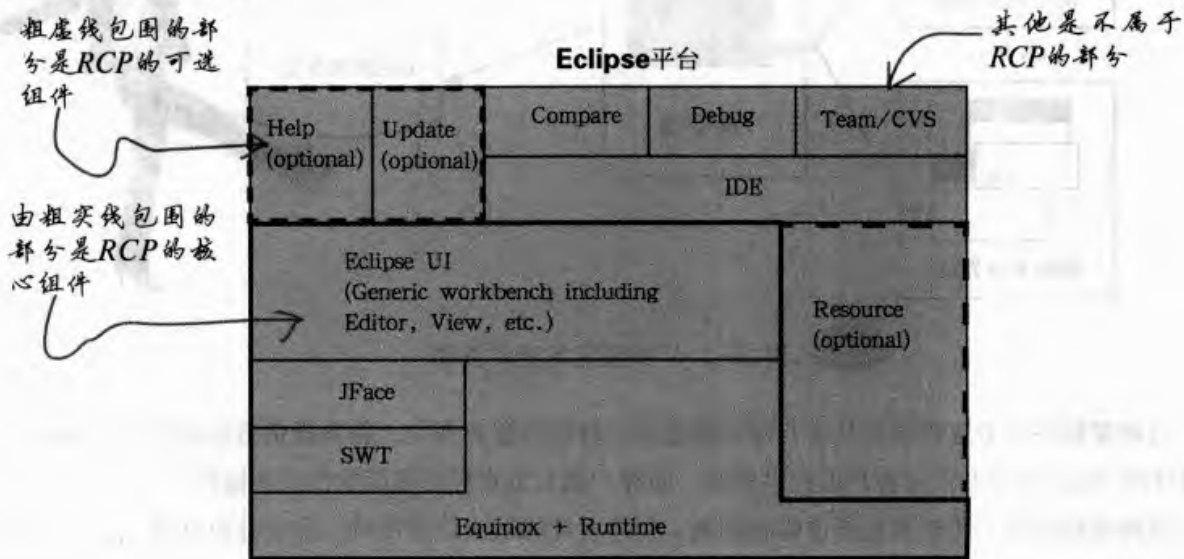


图20-5 Eclipse平台与RCP

由图20-5可见,RCP将调试、版本控制等模块从传统的Eclipse平台中剥离,仅保留了作为核心的运行时框架和以SWT,JFace为基础的UI框架两大部分,同时将帮助系统模块和自动更新模块作为可选项,搭建了一个通用的程序开发平台。

RCP具有成为一个优秀的平台的所有特点,具体如下所示。

- ★插件化的架构使得程序结构更加清晰简练,易于修改和维护;
- ★图形界面技术基于SWT和JFace,所开发出的程序天然跨图形平台,而且具有和本地程序一致的显示效果;
- ★使用Eclipse的UI组件(透视图、编辑器、视图、向导页、帮助系统等)可以轻易设计功能丰富且美观的界面;
- ★拥有强大的Eclipse开源社区支持,开发者可以自由选择数量繁多的各种免费插件无缝集成到程序中以增强程序功能。

最重要的一点是RCP是一个完全免费的平台。无论是商业用户还是个人用户,只要遵循EPL,都可以合法地将其用于包括商业软件开发在内的各种目的。目前,基于RCP的软件产品正在迅猛发展起来。

20.2 RCP平台架构

在学习RCP程序的开发之前，必须对RCP平台有一个基础的了解。本节将介绍RCP平台相关的基础知识，这包括RCP平台的组成和RCP程序的结构两部分内容，另外本节也将对RCP程序的启动过程做简要的介绍和分析。

20.2.1 RCP结构概述

Eclipse RCP平台主要由三大部分构成，如图20-6所示。

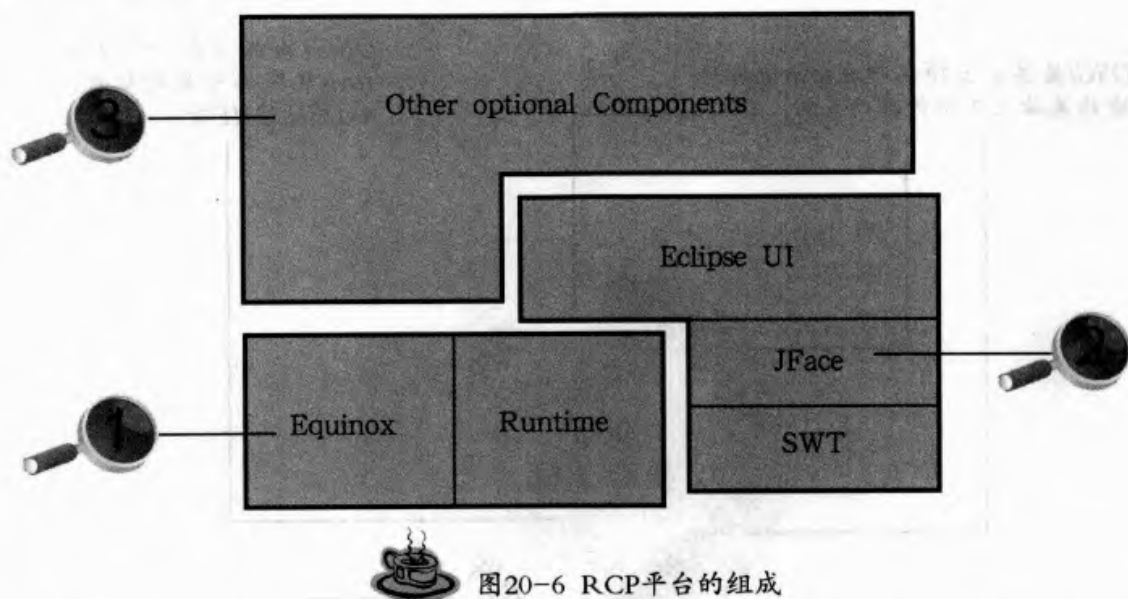


图20-6 RCP平台的组成

第1部分由Equinox和Runtime组成，负责管理整个插件结构体系。其中Eclipse Runtime模块负责解析和管理插件之间的扩展点和扩展的相互关联；而Equinox则管理插件的版本，依赖关系以及动态的载入、卸载等内容。

在早期的Eclipse设计中，是没有Equinox这一模块的。那时，两部分工作都由Eclipse Runtime模块完成。然而，当时的Eclipse平台不能支持动态的载入或卸载插件功能。安装新的插件或进行自动升级后，都必须重新启动工作台才能使新安装的部分生效。

自3.0开始，平台的开发者将目光转向了OSGi框架。OSGi(Open Service Gateway initiative, 开放式服务网关协议)框架是一套基于Java的动态组件模型管理框架。它是由IBM, Apache等数家机构所组成的OSGi联盟所提出的一套开放式标准。J2SE规范中缺乏对组件编程的支持，这套标准恰好对此做了补充。

OSGi框架着眼于组件化编程，为传统的Java执行环境提供了三个方面的服务支持，如图20-7所示。

★模块层(Modules)。OSGi框架对基于J2SE的类型读取机制(Class loading policy)进行了加强，加入了模块化的概念。在传统的Java类型读取机制中，一个程序通常拥有一个单一的、包含所有类型

和资源的类型读取器,模块层则为每个模块提供了独立的类读取器,这意味着用户可以创建只在模块内可见的类、对象或资源,模块层还为模块之间的访问提供了完善的控制机制。

★生命周期管理(Life Cycle)。生命周期管理层负责在运行时对组件进行管理,通过这一层提供的服务,开发者可以动态地添加、删除、升级、启动或禁用各个组件。

★服务注册表(Service Registry)。服务注册表帮助动态变化的组件维护互相引用的关系。组件之间经常需要互相交互,如果通过传统的做法——共享类型和对象来完成这种交互,在其中一方的组件被动态删除或禁用时可能会遇到问题,服务注册表提供了一个综合化的模型用来完成这项工作,在服务注册表中,每一个需要共享的Java对象都是一项服务(Service),当某一项服务可用或不可用时,服务注册表都会发出对应的事件,组件可以通过这些事件了解服务的变化并相应的做出反应。

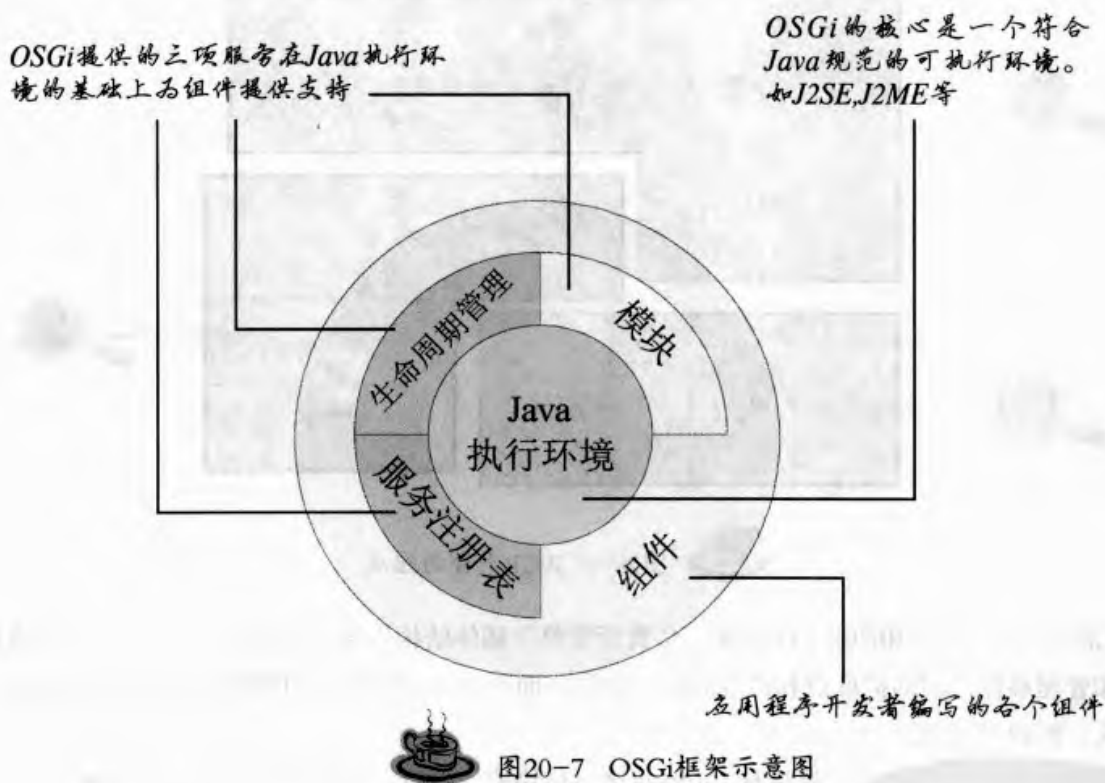


图20-7 OSGi框架示意图

图20-7说明,在OSGi框架下,应用程序不再是完整的一个集合,而由多个组件构成,这些组件拥有各自的生命周期,独立地完成应用程序的一部分工作,应用程序需要调用OSGi框架的服务功能以组织和管理这些组件,这与Eclipse平台的插件思想是相似的,基于这种相似性,Eclipse的开发者开始尝试把OSGi引入插件体系。

自3.0开始,开发者们基于OSGi 框架对旧的插件管理部分做了重构,所形成的新的插件管理框架被称为Equinox,它遵循OSGi Release 4规范。在Equinox中,组件(物理上通常表现为一个JAR文件)被称为“束”(Bundle),插件、段及功能部件等都是束,合法的束需要按照Equinox所要求的格式提供描述其自身的信息,对于JAR文件来说,这些信息一般存储在压缩包中META-INF目录下的MANIFEST.MF文件中,它们与组件的XML描述(插件的plugin.xml段的fragment.xml以及功能

部件的feature.xml等)一起构成了一个完整的Eclipse插件描述,图20-8以一个插件为例对此进行了演示。

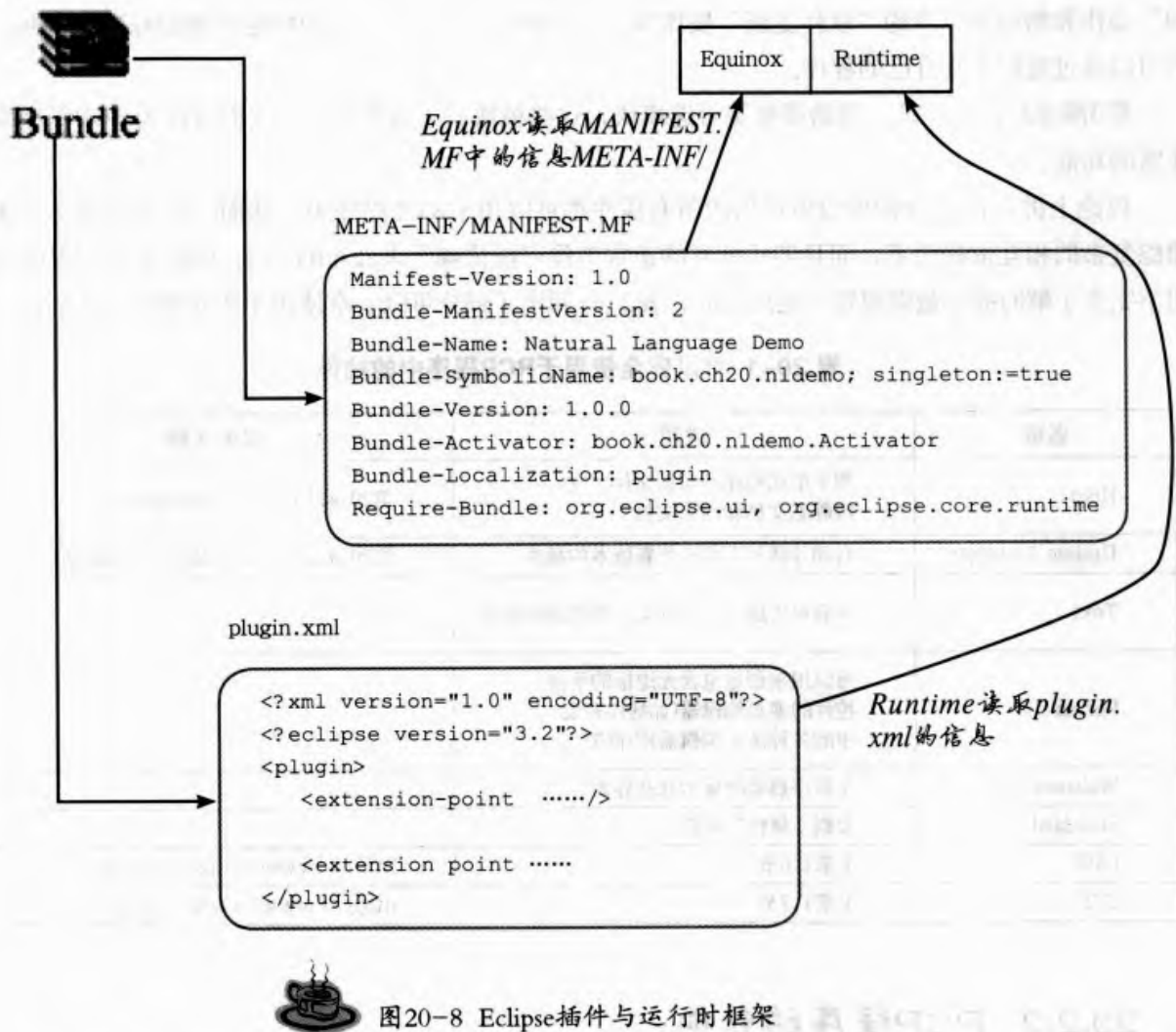


图20-8 Eclipse插件与运行时框架

由图20-8可见,在新的框架中,对一个插件的分析过程是由Equinox共同完成的,Equinox读取MANIFEST.MF文件以确定插件的版本号、相互依赖关系等内容,而Runtime则负责读取plugin.xml并对插件的扩展点、扩展等内容进行维护和管理,两者是相对独立又相互协调的关系。

第2部分是UI相关的插件,由SWT和JFace,以及基于其上的Eclipse UI框架构成。Eclipse UI框架的插件包含了透视图、编辑器、视图和操作集等扩展点,用户在使用Eclipse IDE的时候会发现,视图窗口可以被自由拖曳到窗口的任意边缘,而编辑器区域则始终在中间部分,用户还可以打开多个编辑器并同时将它们水平或垂直地排列在编辑器区域中做比较,这些都是由Eclipse UI所提供的,在RCP程序中,不需要开发者额外的工作,用户就可以享受到这些功能,除此之外,如首选项、向导页等功能也包含在Eclipse UI框架中,在RCP程序中也可以使用这些功能。

在普通的插件开发过程中,工作台窗口是由Eclipse平台提供的,用户只需要开发和管理自己的编辑器、视图等组件,而在RCP程序开发过程中,工作台窗口也需要由用户自行生成,除了这一点之外,为RCP开发图形界面与为插件开发图形界面的工作是完全相同的,用户也可以很方便地利用RCP

将已开发的插件改造成RCP程序。

另外，用户也可以在RCP程序中使用工作台的一些标准的操作，如“窗口”菜单中的“首选项”操作和帮助菜单中的“软件更新”操作等，这些操作会触发工作台特定用途的标准对话框，用户可以通过它们丰富自己的程序。

第3部分是自动升级、帮助系统等可选模块，这些模块属于辅助部分，它们可以为RCP程序添加丰富的功能。

理论上讲，在Eclipse平台中可用的所有插件都可以用于RCP程序中，然而，由于插件之间具有错综复杂的相互依赖关系，而且某些插件的正常工作可能依赖于未公布的API，因此在RCP程序中使用不完全了解的插件是需要冒一定风险的，表20-1列出了部分可以安全使用于RCP程序中的插件。

表 20-1 可以安全使用于RCP程序中的插件

名称	说明	相关文档
Help	用于生成程序的帮助文件。支持静态文档和动态文档	第20.4.1节“为RCP添加帮助内容”
Update Manager	自动寻找并升级有更新版本的插件	第20.4.2节“RCP程序的自动更新”
Text	一套可供建立强大的文本编辑器的框架	
Forms	可以用来创建包含无边框的平滑控件的多页编辑器(如插件开发中的各种清单编辑器)的框架	
Welcome Page	在程序启动时显示欢迎界面	
Standard Views	大纲、属性等视图	
EMF	见第1.6节	http://www.eclipse.org/emf
GEF	见第1.7节	http://www.eclipse.org/gef

20.2.2 RCP程序的结构

Eclipse的一切都基于插件，RCP程序也不例外，通常所说的RCP程序开发，事实上就是开发一组插件，并将它和RCP平台一起发布出去，而RCP插件与普通插件的区别，就在于org.eclipse.core.runtime.applications(下面简称为applications)这个扩展点。

applications扩展点是Eclipse平台运行的“入口”，可以将它看成RCP程序的main函数。默认情况下，启动eclipse.exe时，平台会根据{eclipse根目录}/configuration/config.ini文件中设定的eclipse.application属性的值来决定要启动的入口点，可以在这个文件中修改设定的值，也可以在运行eclipse.exe时使用参数-application APP_EXTENSION_ID来启动其他的入口。

Eclipse IDE安装完成后，默认的启动入口点是id为org.eclipse.ui.ide.workbench的applications扩展，它会负责打开一个工作台窗口，普通插件不需要自己的入口点，用户调用这个插件时，工作台窗口会负责将插件载入，然而该默认入口点是IDE插件的一部分，是不包含在RCP平台中的，因此RCP插件必须扩展applications以声明自己的入口点。是否有自己的applications扩展作为入口点，是RCP插件与普通插件的最大区别。图20-9演示了一个RCP程序的启动过程。

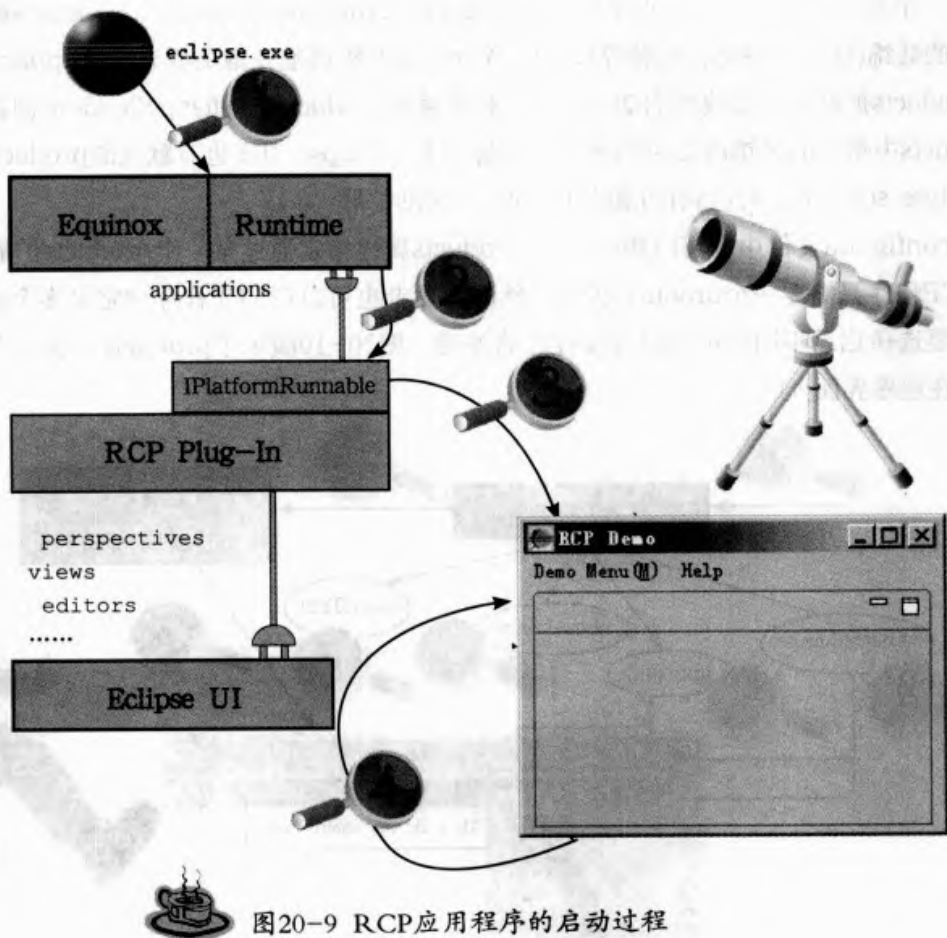


图20-9 RCP应用程序的启动过程

图中标记说明如下：

- 执行eclipse.exe，初始化插件框架；
- 框架通过指定的applications扩展，找到并运行RCP插件中的IPlatformRunnable实例；
- RCP插件在入口点中创建并打开一个工作台窗口，在这个过程中，需要为该工作台窗口指定一个初始透视图id。
- 工作台窗口启动后，会根据指定的初始透视图id启动透视图，透视图又会启动其中包含的视图等组件，这样就得到了一个用户可以操作的界面。

在图20-9中，RCP程序的启动经历了4个步骤，在第 步中，如果框架调用的是id为org.eclipse.ui.ide.workbench的applications扩展，启动的将是Eclipse IDE；而如果调用的是RCP程序提供的applications扩展，启动的就是开发者自己编写的RCP程序了。因此从根本上来说，Eclipse IDE也只不过是一个比较复杂的RCP程序而已。

在图20-9中所示的4个步骤中，平台完成了前面两步，因此编写RCP程序的主要工作集中在 两步上，在第20.3.1节“创建RCP工程”中，将详细讲述RCP程序启动工作台窗口，并显示界面的过程。

另一个与启动相关的扩展点是`org.eclipse.core.runtime.products`，`products`扩展点用于指定对工作台的装饰(如窗口图标、标题等)，每一个`products`扩展唯一地关联着一个`applications`扩展，因此通过`products`扩展也可以找到启动入口点，如果通过`products`启动的`applications`创建了工作台，那么在`products`中指定的装饰就会应用到工作台窗口上。Eclipse IDE也有默认的`products`扩展，它的id是`org.eclipse.sdk.ide`，对应到IDE默认的`applications`扩展。

在`config.ini`文件中，可以用`eclipse.products`属性指定通过某一个`products`扩展启动程序，通常一个RCP程序只需要一个`products`扩展，然而开发者也可以在RCP程序中定义多个`products`扩展，并根据需要选择启动不同的扩展以改变程序的外观，图20-10演示了`products`扩展点中的部分属性是如何体现在程序界面上的。

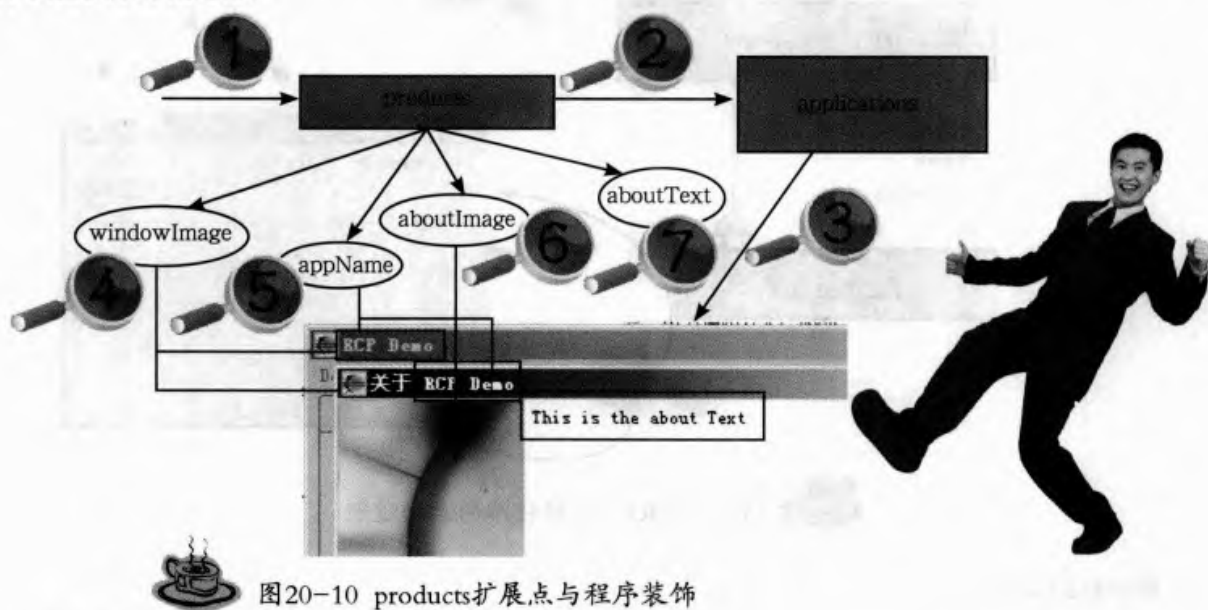


图20-10 products扩展点与程序装饰

图中标记说明如下：

- 1 通过`products`扩展启动平台；
- 2 `products`扩展调用它所对应的`applications`扩展；
- 3 `applications`扩展创建一个工作台窗口；
- 4 `products`扩展的`windowImage`属性可以用来指定工作台窗口的图标；
- 5 `products`扩展的`appName`属性可以用来指定工作台窗口的标题；
- 6 `products`扩展的`aboutImage`属性用于指定显示在“关于...”对话框中的图片；
- 7 `products`扩展的`aboutText`属性用于指定显示在“关于...”对话框中的文字。

从图20-10中可以看出，`products`扩展事实上是由多个属性-值对组成的集合，每个属性都对应

着界面上的不同内容，下面列出了一些常用的属性及它们的作用。

aboutText文本。可以包含程序的版权说明、法律声明等内容，文本内容最长为15行，每行75个字符，可以使用“\n”来换行，这里指定的文本信息会显示在“关于”对话框中。

featureImage指定一个32×32像素的图片，这个图片将显示在“关于”对话框中。

windowImage指定一个16×16像素的图片作为工作台窗口的主图标，图片的路径是以插件为参照的相对路径。

aboutImage显示在“关于”对话框中的图片，这个属性和aboutText属性一起构成了“关于”对话框的主要内容。图片尺寸最大为500×330像素，但为了给aboutText的显示留下足够的空间，图片的尺寸不能超过250×330像素。超过这个范围时，“关于”对话框中会只显示图片，文字将被隐藏。

从Eclipse 3.1开始，Eclipse IDE提供了一个更为简单高效的方法供开发者配置并生成RCP程序的品牌配置，即产品配置文件（Product Config）。通过产品配置文件编辑器，开发者不再需要手动向product扩展点添加属性，可以使用图形化的界面方便地配置这些信息，其界面如图20-11所示。

启动屏幕

当产品启动时就会出现启动屏幕。如果未指定它的位置，则认为“splash.bmp”文件位于产品的定义插件中。

指定启动屏幕所在的插件：

插件： 浏览...

☐ 添加进度条

X 偏移： Y 偏移： 宽度： 高度：

☐ 添加进度消息

X 偏移： Y 偏移： 宽度： 高度： 文本颜色：

窗口图像

指定将与应用程序窗口相关联的图像。这些 GIF 图像通常位于产品的定义插件中。

16x16 图像： 浏览...

32x32 图像： 浏览...

48x48 图像： 浏览...

64x64 图像： 浏览...

128x128 图像： 浏览...

关于对话框

定制“关于”对话框的文本和图像。GIF 图像通常位于产品的定义插件中，其大小不能超过 500x330 像素。如果图像大小超过了 250x330 像素，则不会显示文本。

图像： 浏览...

文本：

欢迎页面

第一次启动产品时将显示欢迎页面。这是为了向新用户介绍产品的功能部件。

指定带有此产品品牌的欢迎页面：

简介标识： 新建...



图20-11 使用产品配置界面配置品牌信息

20.3 开发RCP产品

本节将着手创建一个RCP程序，并将它发布成独立的可执行程序。只要客户的机器上装有JRE，就可以运行它。

20.3.1 创建RCP工程

选择“文件”→“新建”→“项目”菜单，打开新建项目对话框，在插件项目类别中，选择插件项目，并为插件取名“rcpdev.rcpdemo”，在创建向导的第二个页面“插件内容”中，将“富客户机应用程序”选项设为“是”以创建一个RCP程序，如图20-12所示。

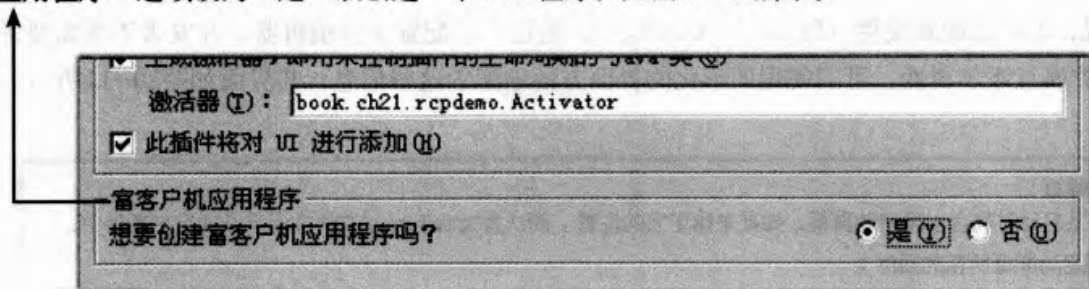


图 20-12 创建富客户机应用程序

在向导的下一页“模板”中，需要选择一个模板来创建RCP程序，这里选择最简单的Hello RCP模板，所有设置保持默认，单击“完成”按钮生成RCP项目。

向导生成的这个RCP程序不需要任何修改，就可以运行。现在，在插件清单编辑器的“概述”页面中单击“启动Eclipse应用程序”，运行结果如图20-13所示。

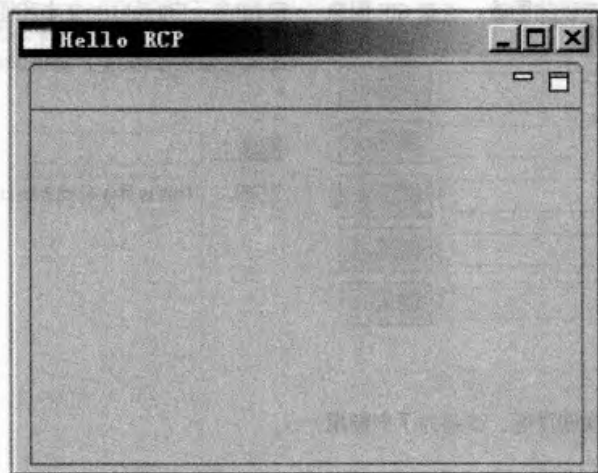


图20-13 第一个RCP程序的运行结果

这个窗口就是由RCP程序产生的工作台窗口，创建它的代码是由向导自动生成的，在创建RCP程序时，向导自动生成了如图20-14所示的文件，并且生成了两个扩展。

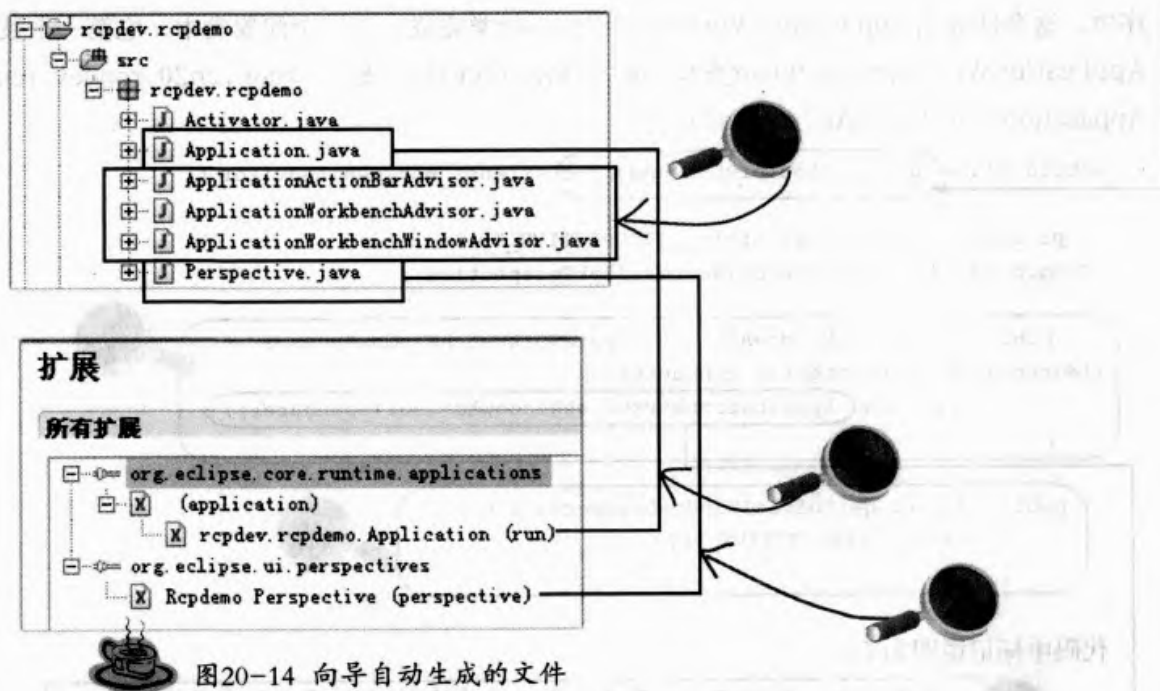





图20-14 向导自动生成的文件

图中标记说明如下:

-  RCP应用程序的入口点, 实现了IPlatformRunnable接口。
-  perspectives扩展点的一个扩展, 为RCP程序提供了默认的透视图。
-  这三个配置类用于配置RCP程序所生成的工作台窗口。

由图20-14可见, applications扩展中将Application类指明为启动入口, RCP程序启动后, 会调用该类的run方法, 为了了解RCP程序的工作原理, 有必要对向导生成的代码, 特别是三个用于配置工作台的类作简单的分析。Application类代码如下所示(代码见光盘: \book.ch20.rcpdev.rcpdemo.Application.java)。

```
public class Application implements IPlatformRunnable {

    public Object run(Object args) throws Exception {
        Display display = PlatformUI.createDisplay();
        try {
            int returnCode =
                PlatformUI.createAndRunWorkbench(display, new
                ApplicationWorkbenchAdvisor());
            .....
        }
    }
}
```

程序在入口处调用了启动工作台的代码, 并提供了一个工作台配置类的实例

在入口点方法中, 代码调用了PlatformUI的静态方法createAndRunWorkbench以创建一个工作台, 这个方法需要一个WorkbenchAdvisor的对象来对所生成的工作台进行配置, 在示例程

序中, 这个任务由ApplicationWorkbenchAdvisor类完成, 在三个配置类中, 它第一个被调用。ApplicationWorkbenchAdvisor类代码如下所示(代码见光盘: \book.ch20.rcpdev.rcpdemo.ApplicationWorkbenchAdvisor.java)。

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {

    protected static final String PERSPECTIVE_ID =
        "com.plugindev.addressbook.AddressBookPerspective";

    public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor
        (IWorkbenchWindowConfigurer configurer) {
        return new ApplicationWorkbenchWindowAdvisor(configurer);
    }

    public String getInitialWindowPerspectiveId() {
        return PERSPECTIVE_ID;
    }
}
```

代码中标记说明如下:

调用一个工作台窗口配置类完成工作台窗口的配置(如是否显示菜单栏, 是否显示工具栏等)。

为工作台窗口配置默认的透视图ID, 当工作台窗口打开时, 会启动这里指定的透视图。在本例中, 所指定的透视图ID对应着生成的Perspective类(代码见rcpdev.rcpdemo.Perspective.java)。

在工作台配置类中, 会创建一个工作台窗口配置类以对将要产生的工作台窗口进行配置, 工作台窗口配置类继承自WorkbenchWindowAdvisor类, 在其中可以配置窗口标题、是否显示菜单栏、工具栏等与窗口相关的内容。除此之外, 工作台窗口配置类还要提供一个操作配置类, 用于为窗口初始化工具栏(工作台窗口中的工具栏是CoolBar)和菜单内容。

ApplicationWorkbenchWindowAdvisor类的代码如下所示(代码见光盘: \book.ch20.rcpdev.rcpdemo.ApplicationWorkbenchWindowAdvisor)。

```
public class ApplicationWorkbenchWindowAdvisor extends
    WorkbenchWindowAdvisor {
    public ApplicationWorkbenchWindowAdvisor
        (IWorkbenchWindowConfigurer configurer) {
        super(configurer);
    }

    public ActionBarAdvisor createActionBarAdvisor
        (IActionBarConfigurer configurer) {
        return new ApplicationActionBarAdvisor(configurer);
    }

    public void preWindowOpen() {
        IWorkbenchWindowConfigurer configurer =
            getWindowConfigurer();
        configurer.setInitialSize(new Point(400, 300));
        configurer.setShowCoolBar(true);
        configurer.setShowMenuBar(true);
        configurer.setShowStatusLine(false);
    }
}
```

代码中标记说明如下:

设置窗口初始尺寸, 是否显示工具栏, 是否显示菜单栏等各种信息。

指定一个操作配置类的实例。

将默认的不显示工具栏改为显示工具栏。

在preWindowOpen()中, 工具栏 (CoolBar) 被设置成不显示, 为了显示工具栏, 要调用IWorkbenchWindowConfigurer.setShowCoolBar方法。

窗口配置类中的preWindowOpen是一个回调方法, 在打开窗口之前, 这个方法会被调用, 为了方便开发者在窗口启动的不同阶段对窗口进行操作, 窗口配置类中提供了多个这样的回调方法, 开发者可以重载它们, 常用的回调方法如表20-2所示(按打开窗口时方法被调用的先后顺序列出)。

表20-2 窗口配置类中的回调方法

方法名	说明
postWindowCreate	在创建窗口后,但在窗口显示前被调用。这时窗口的Shell实例已经被创建
preWindowOpen	显示窗口前被调用。这时窗口的Shell已经被创建, 但Shell中的控件还未被创建。通常在这个方法中配置是否显示工具栏、菜单栏等
postWindowOpen	显示窗口后调用。这时各种控件已经被创建出来。可以在这里设置窗口的初始状态, 如最大化、最小化等
preWindowShellClose	在用户单击窗口的关闭按钮时被调用。开发者可以在这里判断窗口的状态并询问用户是否真的要关闭窗口
postWindowClose	当窗口被关闭后调用, 通常执行完这个方法后, RCP程序就退出了

下面的代码通过重载postWindowOpen方法实现了启动RCP程序后将窗口最大化的功能(代码见光盘: \book.ch20.rcpdev.rcpdemo.ApplicationWorkbenchWindowAdvisor)。

```
public void postWindowOpen() {
```

```
    IWorkbenchWindowConfigurer configurator = getWindowConfigurer();
```

```
    configurator.getWindow().getShell().setMaximized(true);
```

```
}
```

代码中标记说明如下:

通过getWindowConfigurer()方法取得一个IWorkbenchWindowConfigurer对象, 所有对工作台窗口的操作都要通过这个对象的getWindow方法取得IWorkbenchWindow来完成。

通过IWorkbenchWindow.getShell()方法取得工作台窗口对应的Shell并使其最大化。

在三个配置类中, 最后被调用的是操作配置类ApplicationActionBarAdvisor, 它负责向工作台窗口的菜单栏、CoolBar和状态栏上添加操作, 开发者可以重载fillMenuBar, fillCoolBar

和fillStatusLine三个方法来完成这一工作，在示例程序中（代码见光盘：\book.ch20.rcpdev.rcpdemo.ApplicationActionBarAdvisor），向菜单栏上添加了一个打开“关于”对话框的操作。

```
public class ApplicationActionBarAdvisor extends ActionBarAdvisor {
    .....
    private IWorkbenchAction aboutAction;

    protected void makeActions(IWorkbenchWindow window) {
        aboutAction = ActionFactory.ABOUT.create(window);
        register(aboutAction);
    }

    protected void fillMenuBar(IMenuManager menuBar) {
        menuBar.add(aboutAction);
    }

    protected void fillCoolBar(ICoolBarManager coolBar) {}
}
```

代码中标记说明如下：

重载父类的makeActions方法，并在该方法中生成操作对象。

在这些配置方法中，可以将makeActions方法中创建的操作添加到菜单、CoolBar和状态栏的贡献管理器上，不过出于灵活性考虑，通常推荐使用actionSet扩展点创建操作，而不是直接硬编码在这里，在稍后的RCP开发实例中，将演示如何配合使用两种方式共同配置操作。

经过这样的分析和修改，现在再次启动该RCP程序，效果如图20-15所示。

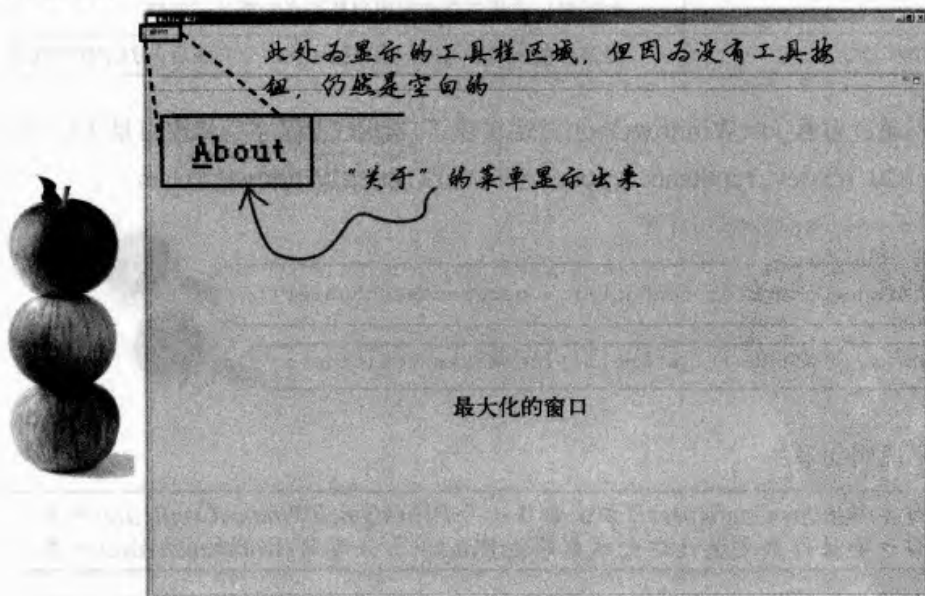


图20-15 修改后的示例RCP程序窗口

了解了RCP程序的构造，下一步的工作就是向程序中添加视图等UI元素，在RCP程序中进行这

些工作和开发插件时完全没有区别。开发者可以开发自己的UI组件，也可以按照自己的喜好使用控制台、属性等系统视图，在示例代码中（代码见光盘：\book.ch20.rcpdev.rcpdemo.Perspective），开发了一个“示例视图” SampleView来显示一行文字，在views扩展点注册了对应的扩展后，将它加入了RCP程序的透视图Perspective类中，显示效果如图20-16所示。

```
public class Perspective implements IPerspectiveFactory {
    public void createInitialLayout(IPageLayout layout) {
        layout.addView(SampleView.ID, SWT.LEFT, 0.2f, layout.getEditorArea());
    }
}
```

在RCP程序的透视图中添加视图的代码与开发插件时使用的方法是一样的

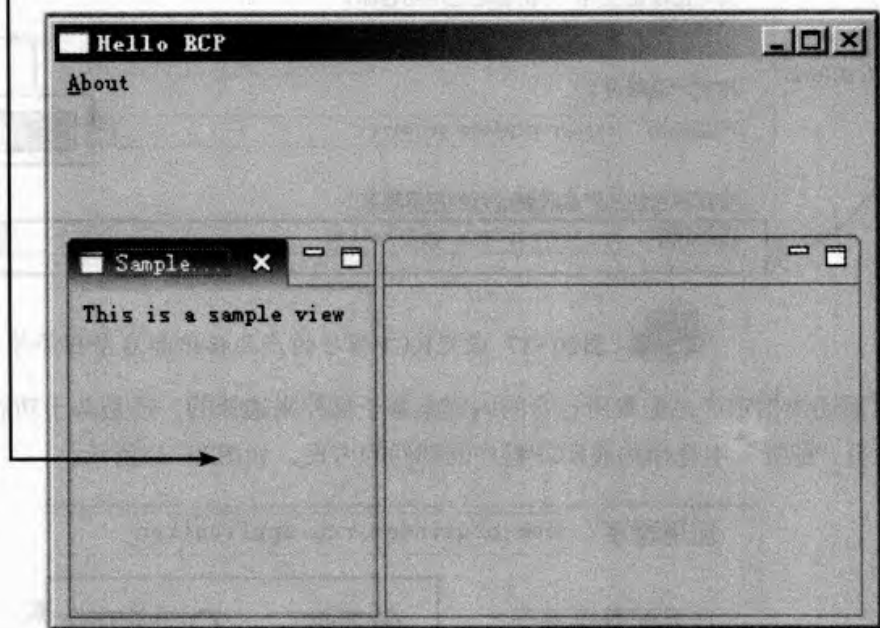


图20-16 向示例的透视图中添加一个视图

※ 注意 : ※

在RCP程序中，可以使用Eclipse环境中所有插件。如果开发者愿意，完全可以将IDE的各种功能添加到自己的RCP程序中，如在自己的程序中包含Java开发环境等。然而，这种做法是不提倡的。通常的观点认为，RCP程序应该只使用由Eclipse UI所提供的基本UI功能以及平台中相对独立的功能模块（如帮助模块，升级管理模块等），而不提倡使用如备忘单(Cheat Sheet)、资源视图等与平台紧密相关的功能。如果开发者一定需要这些由平台或IDE所提供的功能，可以考虑开发一个插件，然后将它与Eclipse环境捆绑在一起发布出去。



20.3.2 发布RCP工程

本节将讲述如何将上一节开发的RCP程序发布成一个可以脱离Eclipse开发环境独立运行的程序。发布RCP工程之前，首先需要创建一个产品配置文件。打开“新建”对话框，选择“插件开发”

→“产品配置”以新建一个产品配置文件，产品配置必须以.product为扩展名，此处将示例程序的产品配置命名为new.product，完成后，系统会调用产品配置编辑器打开该文件。

在产品配置编辑器的“描述”选项卡的“概述”段中，可以指定所发布RCP程序的名称，启动插件的入口点应用程序，以及产品标识等内容，如图20-17所示。产品标识必须是一个products扩展点的扩展，如果插件中还没有包含一个products扩展，可以单击“新建”按钮以新建一个。



图20-17 设定RCP程序的产品标识和启动程序等

在这个页面还要指定产品配置所包含的内容是基于插件来选择的，还是基于功能部件选择的，这将影响到下一页“配置”中选择构成RCP程序的部件的方式，如图20-18所示。

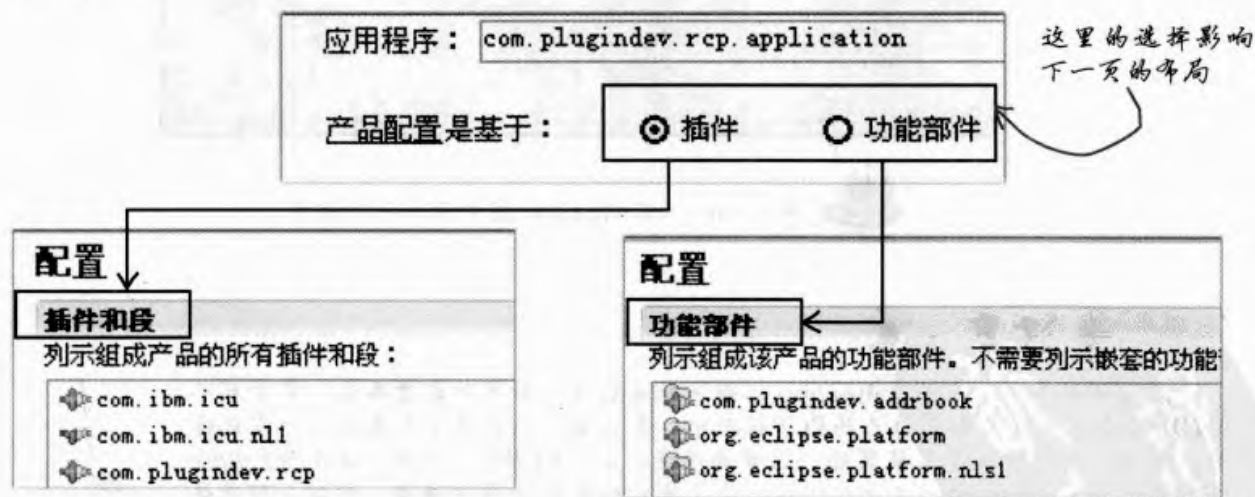
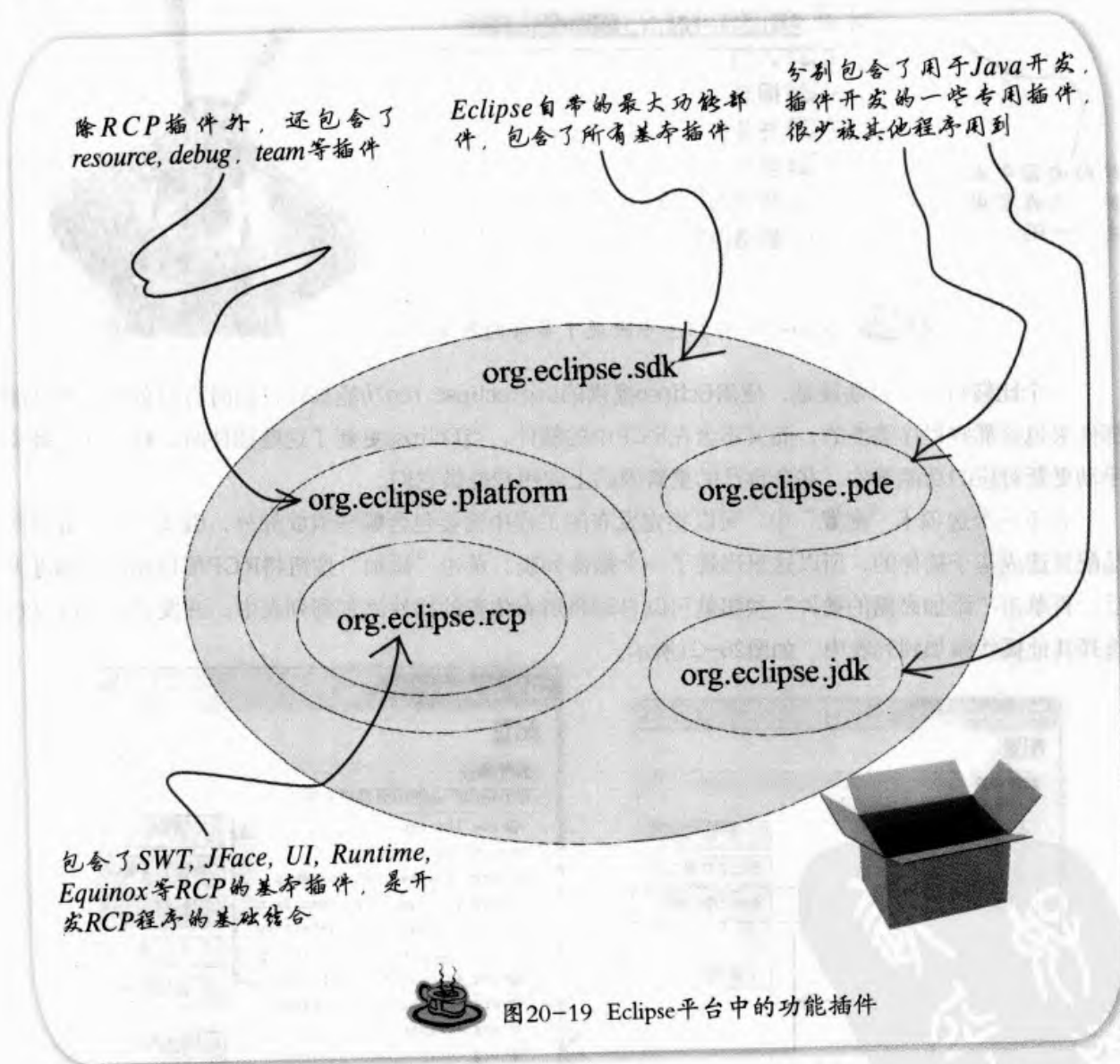


图20-18 选择产品配置是基于插件或功能部件

如果在这里选择了“插件”，那么“配置”选项卡中将会要求开发者选择需要包含在RCP程序中的插件和段，如果选择“功能部件”，那么“配置”选项卡中的选择将会基于功能部件做出，由于此处尚未给RCP插件创建功能部件，因此选择了“插件”。

然而推荐的方式是使用功能部件来构建RCP应用程序，它的好处在于只有使用功能部件，才能启

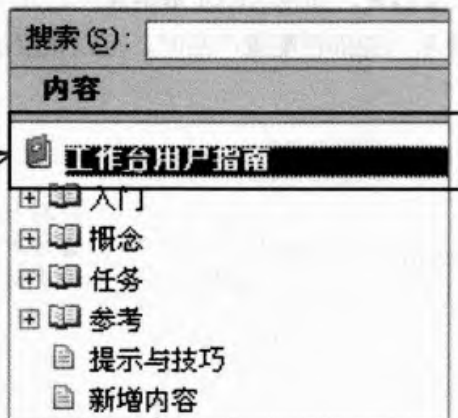
用自动升级功能，这样不仅开发者自己的RCP程序可以升级，当Eclipse社区发布了RCP的新版本时，用户也可以及时得到最新版本；而且使用了功能部件使得程序结构比较清晰，易于管理，因此，虽然使用插件创建产品配置与使用功能部件在功能上没有区别，功能部件仍然是更好的选择。Eclipse将常用的插件都打包为功能部件随平台一起发布出来，如包含RCP基本插件的org.eclipse.rcp，包含平台基本插件的org.eclipse.platform等，在使用功能部件配置产品时，可以直接使用它们，图20-19显示了这些功能部件。



然而，使用Eclipse所提供的默认功能部件组合，最大的问题就是灵活性不够，如为了构建HTML风格的页面，在地址本插件中使用了org.eclipse.ui.forms插件所提供的功能，这个插件不属于org.eclipse.rcp功能部件，而是处于org.eclipse.platform功能部件中的，那么，为了使用这一个插件，就需要将整个platform功能部件导入进来，而其中不需要的插件也会在界面上做出贡献，这可能导致不希望看到的结果，如在菜单栏上多出了和RCP程序完全无关的菜单等，如图20-20所示。

停止本应用程序
 停止 搜索(S) 运行(R) 帮助

使用platform功能部件组成RCP程序时，出现了多余的菜单



帮助内容中也会出现“工作台用户指南”一项



图20-20 在程序中出现了多余的界面元素



一个比较折中的办法就是，使用Eclipse提供的org.eclipse.rcp功能部件，同时自行创建一个功能部件来包含那些程序需要的，而又不含在RCP中的插件，当Eclipse更新了这些插件时，程序开发者要手动更新对应的功能部件，并在自己的更新网站上向用户提供它们。

在下一个选项卡“配置”中，可以指定发布的工程中需要包含哪些组成部分，因为在前一页将产品配置选成基于插件的，所以这里出现了一个插件列表，单击“添加”按钮将RCP项目插件添加进去后，再单击“添加必需的插件”按钮就可以自动将所有依赖的插件添加到列表中。开发者也可以自行选择其他插件添加到列表中，如图20-21所示。



将示例插件添加到列表中后，单击“添加必需的插件”按钮，就可以将其其他需要的插件添加到列表中



图20-21 为RCP程序添加必须的插件

在“启动程序”选项卡中，可以为生成的可执行程序指定名称（如果不指定，默认的名称为 eclipse.exe），以及指定可执行程序的图标和默认启动参数等内容。

最后一个“标记”选项卡就是在第20.2.2节中介绍过的，用于配置程序装饰的页面，它包含了启动屏幕，欢迎页面，窗口图标和“关于”对话框的文字和图片等内容，此处为示例程序指定了“关于”对话框的文字，其他的选项读者可以自行尝试。

设置好上述选项后，在“概述”选项卡中单击“Eclipse产品导出向导”来导出RCP程序，选择导出目录后，就可以将示例RCP程序导出成独立的可执行程序了，如图20-22所示。

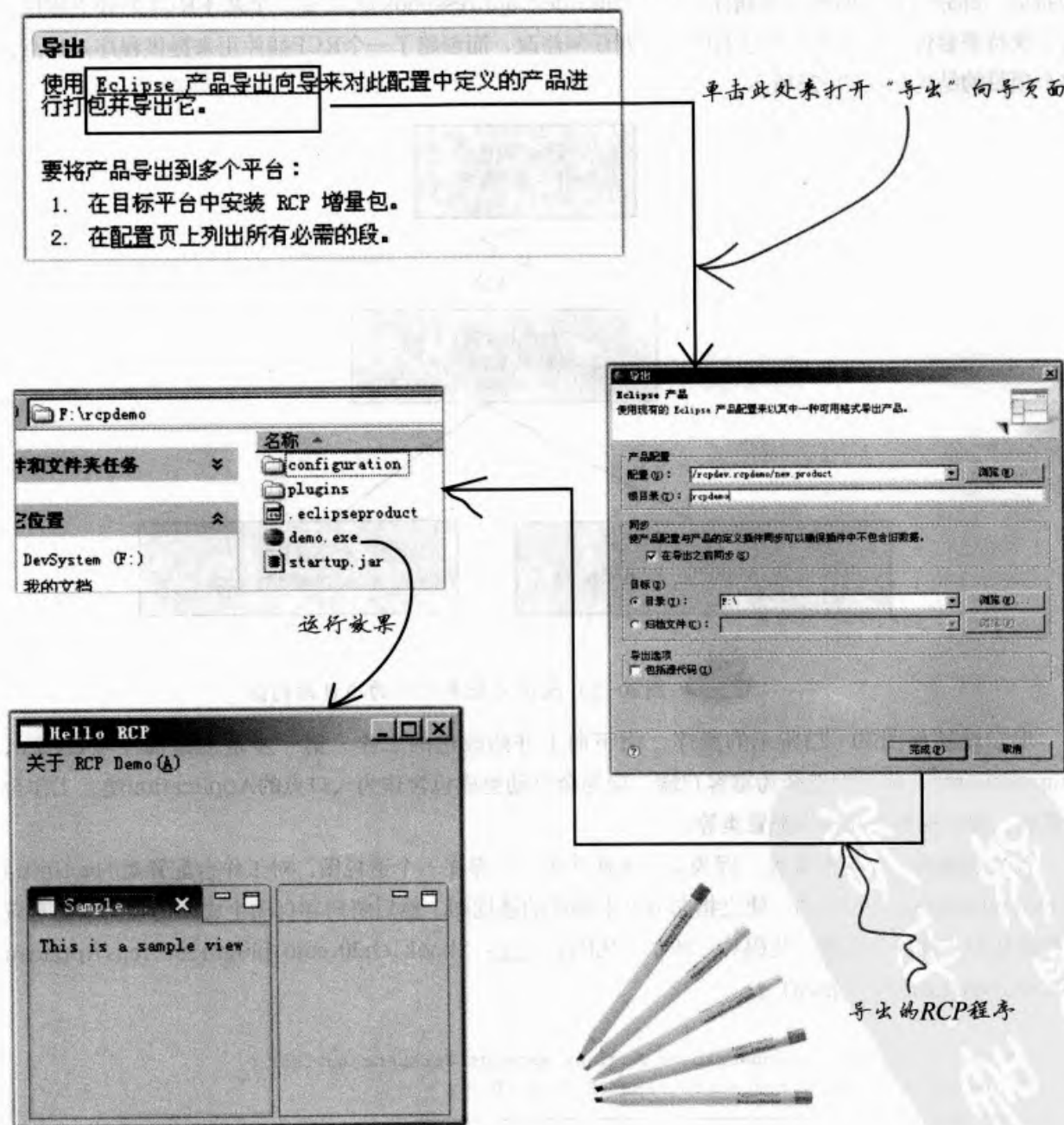


图20-22 将RCP程序导出到文件系统

发布出的RCP程序以一个目录的形式存在，开发者可以手动将其打包成ZIP文件发布，也可以使用第三方的安装程序制作软件(如InstallShield等)将其制作成安装程序发布到最终用户。

20.4 将插件改造成RCP程序

只要了解了RCP程序的原理，将插件改造成RCP程序是一件再容易不过的事情。本节将演示如何将第二部分所开发的地址本插件(com.plugindev.addressbook)改造成一个基于RCP的独立程序。为了保持兼容性，此处将不对已有的插件做任何修改，而创建了一个RCP插件用来提供程序入口点。整个项目的结构如图20-23所示。

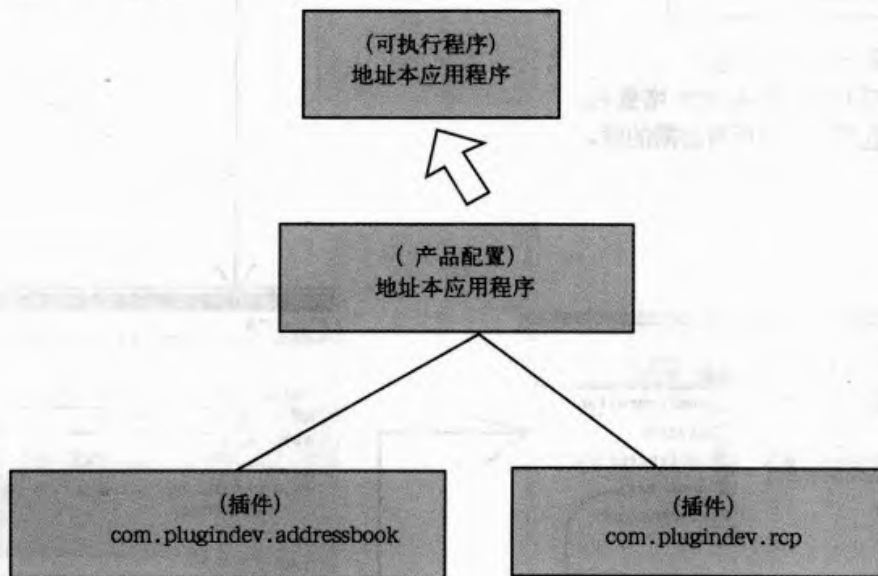


图20-23 改造地址本插件的项目结构图

现在就按照图20-23所示的顺序，由下向上开始改造的工作。第一步是创建插件项目com.plugindev.rcp，将类型选择为富客户端。向导会自动生成包含作为入口点的Application类，工作台配置类，窗口配置类和操作配置类等。

首先要修改工作台配置类，因为地址本插件中已经存在一个透视图，对工作台配置类的getInitialWindowPerspectiveId改动，使之指向地址本插件的透视图，然后将向导自动生成的透视图以及对应的扩展从RCP插件中删除。代码如下所示(代码见光盘：\book.ch20.com.plugindev.rcp.ApplicationWorkbenchAdvisor.java)。

```

public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {
    protected static final String PERSPECTIVE_ID =
        "com.plugindev.addressbook.AddressBookPerspective";

    public String getInitialWindowPerspectiveId() {

```

```

        return PERSPECTIVE_ID;
    }
    .....
}

```

插件com.plugindev.addressbook的plugin.xml

```

<extension
    point="org.eclipse.ui.perspectives">
    <perspective
        class="....."
        icon="....."
    >
        id="com.plugindev.addressbook.AddressBookPerspective"
        name="AddressBook"/>
    </extension>

```

在工作台窗口配置类中，注释掉设置窗口标题的代码，为窗口设置标题的工作将在稍后的产品配置部分完成。如下所示。

```

public class ApplicationWorkbenchWindowAdvisor extends
    WorkbenchWindowAdvisor {

    public void preWindowOpen() {
        IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
        configurer.setInitialSize(new Point(400, 300));
        configurer.setShowCoolBar(false);
        configurer.setShowStatusLine(false);

        // configurer.setTitle("Hello RCP");
    }
}

```

注释掉这段代码

※ 注意 ※

使用代码设置窗口标题和图标等内容的效果与使用产品配置设置它们是完全一样的。但是使用代码设置的话，以后每一次改动就要重新编译整个产品，而在产品配置中设定它们，修改起来就会方便得多，也不需要代码作改动，出于可扩展性的考虑，应该将这些工作交由产品配置完成。

下面需要为RCP产品添加产品配置。在RCP插件中新建一个产品配置default.product，打开它，在“概述”页中，创建一个新的产品标识扩展，命名为product，并选择com.plugindev.rcp.application作为应用程序入口点。

按照第20.3.2节“发布RCP工程”中所讲述的内容对程序进行其他必要的装饰后，在“概述”页中选择“Eclipse产品导出向导”，选择一个目录，将RCP程序导出，现在插件已经被改造成一个可独立运行的程序了，运行效果如图20-24所示。

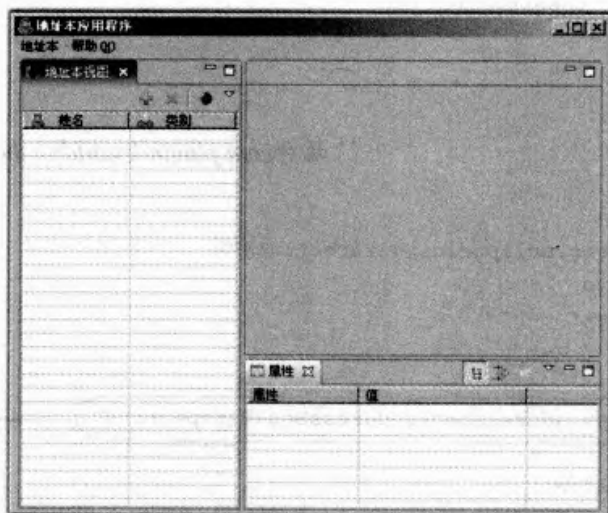


图20-24 独立的地址本应用程序

20.4.1 为RCP添加帮助内容

对于一个软件产品来说，一份完整的帮助内容可以方便用户上手，从而为产品增色不少，本节将演示如何为上面创建的RCP程序添加帮助内容。

首先要向菜单栏中添加打开帮助对话框的操作，修改操作配置类。代码如下所示（代码见光盘：`\book.ch20.com.plugindev.rcp.ApplicationActionBarAdvisor`）。

```
public class ApplicationActionBarAdvisor extends ActionBarAdvisor {
    private IWorkbenchAction helpAction;

    protected void makeActions(IWorkbenchWindow window) {
        helpAction = ActionFactory.HELP_CONTENTS.create(window);
        register(helpAction);
    }

    protected void fillMenuBar(IMenuManager menuBar) {
        IMenuManager menuManager = new MenuManager(
            Messages.getString("ApplicationActionBarAdvisor.helpMenu.label"), "helpMenu");
        //$NON-NLS-1$ //$NON-NLS-2$
        menuManager.add(new GroupMarker("helpContentGroup")); //$NON-NLS-1$
        menuManager.appendToGroup("helpContentGroup", helpAction); //$NON-NLS-1$

        menuBar.add(menuManager);
    }
}
```

代码中标记说明如下：

此处利用Eclipse平台提供的ActionFactory生成一个打开帮助窗口的操作，ActionFactory中还包含了许多用于生成其他系统操作的实例，在创建RCP示例程序时，曾经使用过ActionFactory.ABOUT来创建打开“关于”对话框的操作。



在菜单中创建GroupMarker并将操作添加上去。

现在帮助菜单被添加到了地址本应用程序的菜单中。

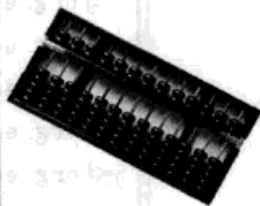
在创建菜单管理器和添加操作时，为所创建的菜单和GroupMarker等都赋了id，这样在通过actionSets扩展点创建操作时，就可以使用这些id所构成的路径来定位，如上面的代码在操作配置类中创建了一个id为“helpMenu”的菜单管理器，并且在其中创建了一个id为“helpContent”的GroupMarker。那么在通过actionSets扩展点创建操作时就可以直接向路径“helpMenu/helpContent”添加一个操作了。

另外，也可以通过向菜单栏顺序添加GroupMarker，然后在创建菜单时指定路径的办法来排列顶级菜单。如向菜单栏中先后添加了GroupMarker “a”和GroupMarker “b”，那么在actionSets中被指定路径为“a”的菜单就会排在路径为“b”的菜单前面。

※ 注意 ※

如果采用“插件”的方式来配置产品，默认情况下，Eclipse自带的帮助系统是不会被添加到RCP程序里面的，这会导致单击“帮助”菜单后没有响应，需要手动往插件列表中添加需要的插件。运行帮助系统所需的插件列表如下：

```
org.eclipse.help.appserver.nl1 (0.0.0)
org.eclipse.help.appserver (0.0.0)
org.eclipse.help.base.nl1 (0.0.0)
org.eclipse.help.base (0.0.0)
org.eclipse.help.nl1 (0.0.0)
org.eclipse.help.ui.nl1 (0.0.0)
org.eclipse.help.ui (0.0.0)
org.eclipse.help.webapp.nl1 (0.0.0)
org.eclipse.help.webapp (0.0.0)
org.eclipse.help (0.0.0)
org.eclipse.tomcat.nl1 (0.0.0)
org.eclipse.tomcat (0.0.0)
```



现在运行地址本应用程序时，“帮助内容”菜单已经出现在“帮助”菜单中了，但由于其中没有内容，单击时会显示“未安装文档”的错误，如图20-25所示。为了向其中添加自定义的帮助内容，需要使用org.eclipse.help系列扩展点。常用的help系列扩展点的相关介绍如表20-3所示。

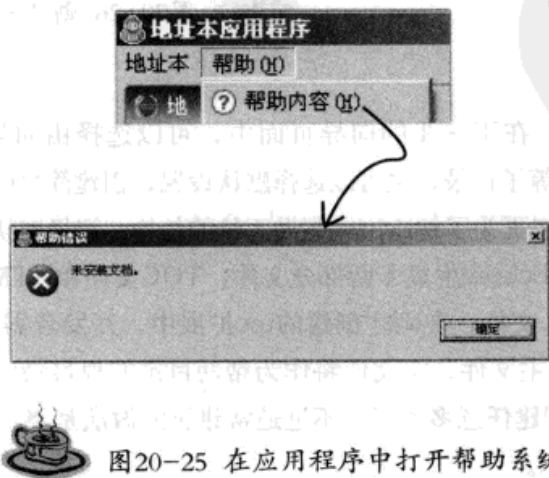


图20-25 在应用程序中打开帮助系统



表20-3 常用的help系列扩展点

扩展点id	扩展点说明
org.eclipse.help.toc	开发者扩展这个扩展点, 以向Eclipse的帮助系统贡献TOC文件。TOC(Table of Content)是特定格式的XML文件, 用于描述帮助系统左侧的帮助文件目录结构。开发者指定帮助文件的目录结构, 并为每个目录项指定一个HTML格式的帮助用户, 就建立了基本的帮助系统
org.eclipse.help.contexts	在插件中提供上下文相关的帮助内容
org.eclipse.help.index	生成帮助文件的索引
org.eclipse.help.contentProducer	用于提供在运行时动态生成的帮助内容

本节将主要介绍如何创建静态的帮助文档以及为其创建索引, 会用到toc及index两个扩展点, 为了创建自己的帮助内容, 首先必须创建一个org.eclipse.help.toc的扩展, 由于help扩展点不属于必须的扩展点, 因此默认情况下, 在扩展点列表中它是被隐藏的, 取消选择“只显示必需插件中的扩展点”项以显示所有扩展点, 就可以找到它了, 如图20-26所示。



图20-26 创建一个toc扩展

选择唯一的可用模版, 在下一步的向导页面中, 可以选择由向导来创建帮助目录中的“入门”、“引用”、“示例”等子目录, 也可以选择默认设置, 创建简单的目录, 然后按照自己的要求添加子目录, 此处为了让读者更为了解Eclipse帮助系统的结构, 选择默认的选项。

单击“完成”按钮后, Eclipse生成了两部分文件, TOC文件和HTML页面文件。TOC文件是描述帮助内容目录结构的XML文件, 在刚才创建的toc扩展中, 开发者需要注册所有使用到的TOC文件, 并指定其中至少一个为主文件, 主文件将作为帮助目录的根目录(一本书)显示在帮助窗口目录内。在一个toc扩展中可以创建任意多本书, 不过通常建议的做法是将一个插件的相关帮助内容集中在一本书中, 如图20-27所示。

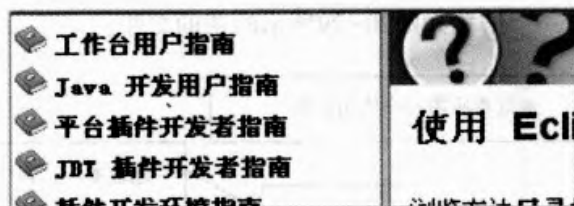


图20-27 Eclipse自带的帮助——每一个插件只有一本书

toc使用topic元素组织和管理目录结构。一个topic元素就是目录树中的一项，其中还指定了这一项所关联的HTML文件名。单击了某一项后，就会显示对应的HTML页面。开发者可以将所有topic元素都写在主toc文件中，也可以将它们分别写在独立的toc文件中，并在主文件中用link元素链接这些toc文件。两种做法的效果是完全一致的，toc文件与目录树之间的关系如图20-28所示。



图20-28 toc文件与帮助文档——如何创建树状结构

下面要为示例程序创建一个结构如图20-29所示的帮助文件。

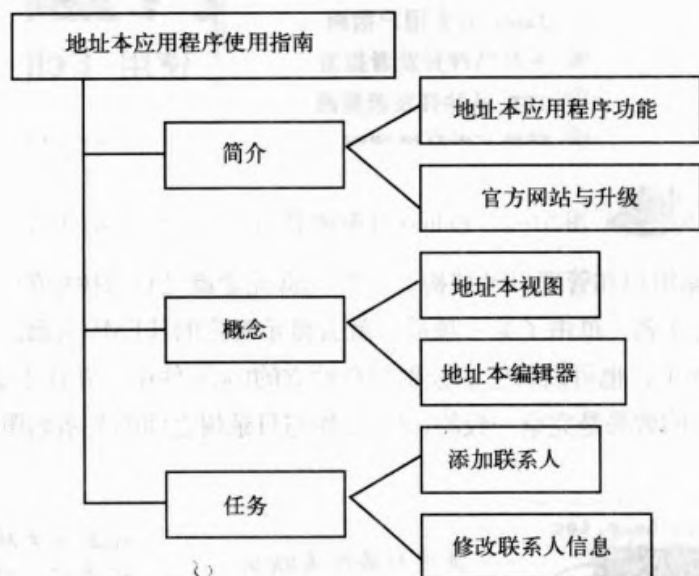


图20-29 帮助文件的结构图

为了方便管理，为每一个子项创建单独的toc文件，相应的HTML文件也分别存放在不同的子文件夹内。

创建的toc文件一共有4个，在toc扩展内将它们注册。代码如下所示。

✦ toc.xml(主toc文件)

```
<toc label="地址本应用程序指南" topic="html/toc.html">
  <link toc="introToc.xml"/>
  <link toc="conceptToc.xml"/>
  <link toc="taskToc.xml"/>
</toc>
```

✦ introToc.xml(子项：简介)

```
<toc label="Intro">
  <topic label="简介" href="html/intro/intro.html">
    <topic label="地址本应用程序功能" href="html/intro/function.html"/>
    <topic label="官方网站与升级" href="html/intro/update.html"/>
  </topic>
</toc>
```

✦ conceptToc.xml(子项：概念)

```
<toc label="概念">
  <topic label="概念" topic="html/concept/concept.html">
    <topic label="地址本视图" href="html/concept/view.html"/>
    <topic label="地址本编辑器" href="html/concept/editor.html"/>
  </topic>
</toc>
```



✦ taskToc.xml(子项: 任务)

```
<toc label="任务">
  <topic label="任务" href="html/task/intro.html">
    <topic label="添加联系人" href="html/task/function.html"/>
    <topic label="修改联系人信息" href="html/task/update.html"/>
  </topic>
</toc>
```

在HTML目录下面创建三个子文件夹concept,intro和task后,将对应的HTML文件放进去。就可以启动帮助系统检查一下成果了,创建好的帮助系统如图20-30所示。

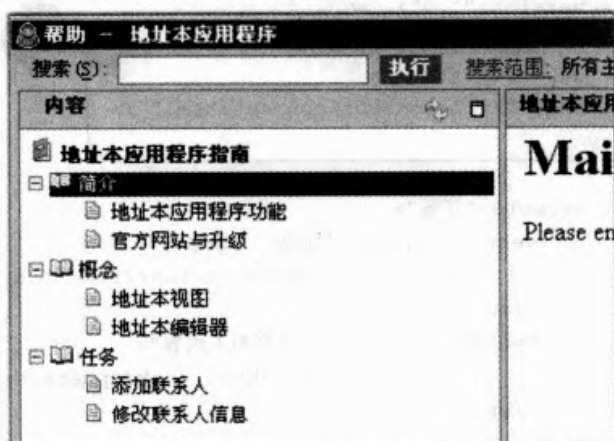


图20-30 创建完成后的帮助系统

下面为帮助系统提供关键字索引。关键字索引与全文检索功能是两个不同的概念,全文检索是Eclipse的帮助系统自动提供的功能,它会搜索所有的HTML文档并找出与搜索目标相近的部分,而关键字索引则是帮助文档的编写者提供的一个类似“术语表”或“FAQ”的东西,编写者可以将术语或关键字组织成一个列表供用户选择,如图20-31所示。

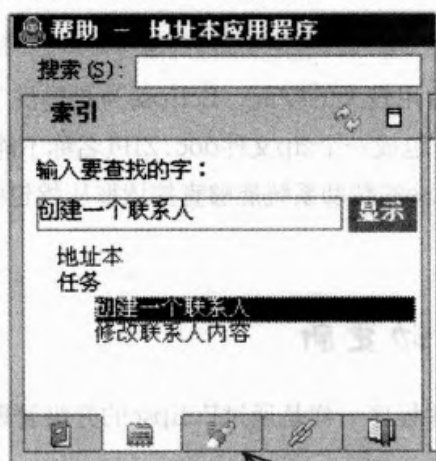
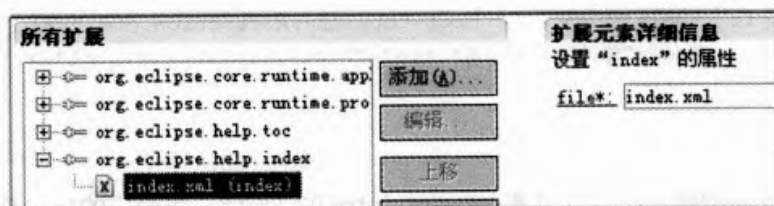


图20-31 关键字索引

为了创建关键字索引,需要使用org.eclipse.help.index扩展点,这个扩展点的内容很简单,只要求开发者提供一个包含了索引内容的XML文件就可以,如图20-32所示的XML文件生成了图20-31中所见的索引内容。



```
<?xml version="1.0" encoding="UTF-8"?>
<index version="1.0">
```

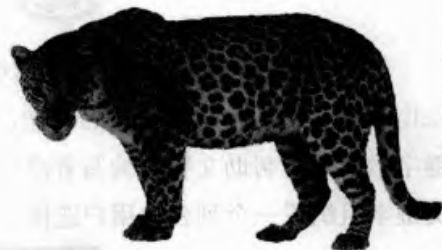
```
<entry keyword="地址本">
  <topic href="html/concept/concept.html" />
</entry>
```

```
<entry keyword="任务">
  <entry keyword="创建一个联系人">
    <topic href="html/task/function.html" />
  </entry>
  <entry keyword="修改联系人内容">
    <topic href="html/task/update.html" />
  </entry>
</entry>
</index>
```

关键字索引的每一项都是一个entry元素,元素可以嵌套。entry元素所包含的topic子元素则指定了单击该项时所要显示的HTML页面



图20-32 创建关键字索引



当帮助文档内容增长起来时,会占用较大的空间。Eclipse为此提供了一个小功能,当程序发布时,开发者可以将所有的HTML文件打包成一个zip文件doc.ZIP(名称不能随便改动),并将这个压缩包放在插件目录中一起发布出去,Eclipse的帮助系统能够直接读取压缩包中的HTML文件,这就方便了帮助内容的管理和发布。

20.4.2 RCP程序的自动更新

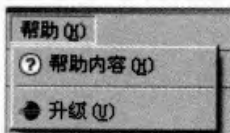
RCP程序的自动更新功能,与插件程序一样是通过Eclipse的升级管理器完成。本节将介绍如何在RCP程序中使用升级管理器。

首先在帮助菜单中添加一个操作,使用户能够启动升级管理器,此处通过actionSets扩展完成这一工作。代码如下所示。

```
<extension
    point="org.eclipse.ui.actionSets">
    <actionSet
        id="com.plugindev.rcp.actionSet"
        label="HelpActionSet"
        visible="true">
        <menu id="helpMenu"
            label="%actionSet.helpMenu.label"
            path="AddressBookMenu">
            <separator name="updateGroup"/>
        </menu>
        <action class="com.plugindev.rcp.actions.UpdateAction"
            icon="icons/sample.gif"
            id="com.plugindev.rcp.actions.UpdateAction"
            label="%actionSet.updateAction.label"
            menubarPath="helpMenu/updateGroup"
            tooltip="%actionSet.updateAction.tooltip"/>
        </actionSet>
    </extension>
```

这里所声明的帮助菜单的id是和前面在ActionBarAdvisor中创建的帮助菜单id一致的。这样就可以将代码中创建的操作和在plugin.xml中声明的操作放置在一个菜单上

这样就将一个“升级”的菜单项添加到了帮助菜单中，如下所示。



为了使用Eclipse平台提供的自动更新功能，将插件org.eclipse.update.ui加到RCP应用程序的依赖列表中，然后在UpdateAction的run方法中，添加如下的代码来调用升级管理器（代码见光盘：`\book.ch20.com.plugindev.rcp.actions.UpdateAction`）。

```
UpdateJob job = new UpdateJob("Search for update", false, false);
```

```
UpdateManagerUI.openInstaller(window.getShell(), job);
PlatformUI.getWorkbench().getProgressService().showInDialog(
    window.getShell(), job);
```

代码中标记说明如下：

启动系统自带的，查找更新内容的Job，这段代码将弹出一个对话框显示“Search for update”并显示一个进度条。

如果找到了更新的内容，启动安装管理器安装它们。

在运行之前，还要将这些插件添加到功能部件中，它们都是运行升级程序所必需的，如下所示。


```

org.eclipse.update.configurator
org.eclipse.update.configurator.nl1
org.eclipse.update.core
org.eclipse.update.core.nl1
org.eclipse.update.core.win32
org.eclipse.update.ui
org.eclipse.update.ui.nl1

```

现在启动地址本应用程序，单击“升级”菜单，就可以自动执行升级操作，如图20-33所示。

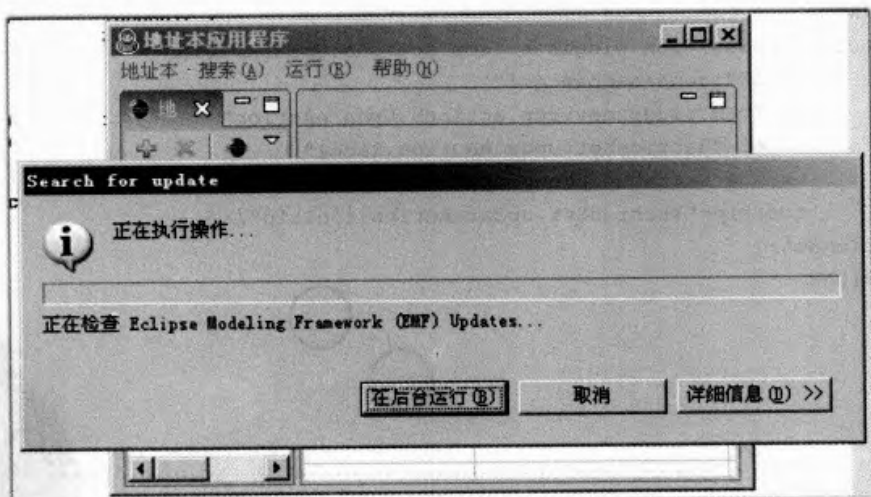


图20-33 使用升级管理器升级RCP程序

20.4.3 为程序添加欢迎页面 (Intro)

Eclipse在第一次启动时，会向用户显示一个华丽的欢迎页面，其中包含了对初学者非常有用的一些介绍和示例程序的链接。如果能为自己的RCP程序添加一个类似的欢迎页面，介绍产品的功能、用法和一些提示，对方便用户使用是很有好处的。本节将介绍如何为RCP程序添加欢迎页面。

为了使用欢迎页面，需要在插件中扩展两个扩展点org.eclipse.ui.intro和org.eclipse.ui.intro.config。开发者并不需要手动创建这两个扩展，可以使用产品配置编辑器提供的功能方便地生成它们，打开产品配置编辑器的“标记”选项卡，找到“欢迎页面”一项，单击“新建”按钮即可进行创建，如图20-34所示。

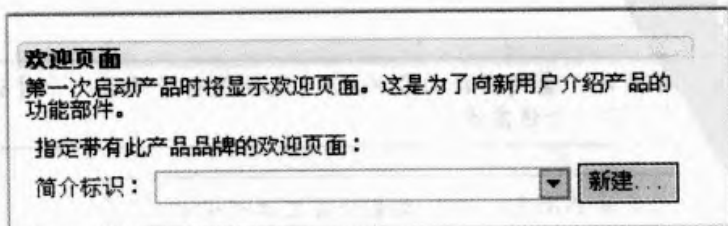


图20-34 新建一个欢迎页面

在出现的向导对话框中，选择欢迎页面所在的目标插件，向导会自动生成所需的两个扩展，如图20-35所示。

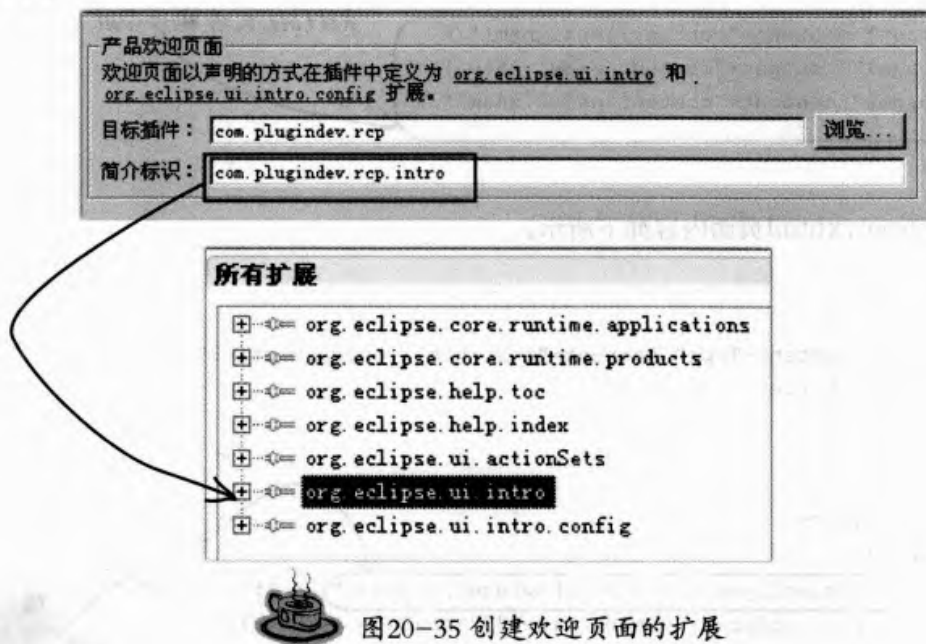
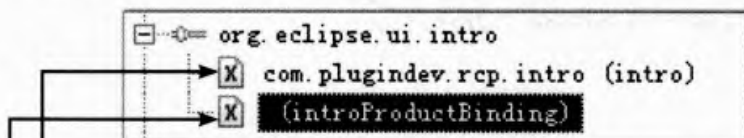


图20-35 创建欢迎页面的扩展

生成的intro扩展中包含了两个配置选项，如下所示。



在第一个选项中，需要指定一个org.eclipse.ui.IntroPart接口的实现类。如果需要编写自己的实现，可以继承org.eclipse.ui.IntroPart抽象类，本示例中使用Eclipse提供的默认实现org.eclipse.ui.intro.config.CustomizableIntroPart类，这个类允许使用者提供符合规定格式的XHTML文件作为欢迎页面的内容。

第二个选项则将Intro扩展绑定到一个products扩展上。当程序从某个product启动时，对应的intro将被载入。

另外一个扩展org.eclipse.ui.intro.config就是用于向CustomizableIntroPart提供XHTML页面内容的，为了方便说明，直接看plugin.xml的内容。如下所示。

```
<extension point="org.eclipse.ui.intro.config">
  <config content="introContent.xml"
    id="com.plugindev.rcp.introConfigId"
    introId="com.plugindev.rcp.intro">
    <presentation home-page-id="root">
      <implementation kind="html"/>
    </presentation>
  </config>
</extension>
```

在config扩展中，首先指定了一个包含所有使用到页面信息的XML文件，可以把它看成页面信息的注册表

指定作为根页面的文件id，这个id也是在上面的XML文件中指定的

作为“注册表”的introContent.xml的内容如下所示，所有将要显示在欢迎页面中的XHTML文

件都必须在这个文件中用一个page标签声明并且赋一个唯一的id来标识。

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<introContent>
```

```
<page id="root" content="content/root.xhtml"/>
<page id="page1" content="content/page1.xhtml"/>
<page id="page2" content="content/page2.xhtml"/>
```

在这里需要为所有使用到的XHTML文件赋一个id

```
</introContent>
```

而作为根页面的root.xhtml页面内容如下所示。

```
<html>
```

```
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

```
<title>root.xhtml</title>
```

```
</head>
```

```
<body>
```

```
<h1>Address Book</h1>
```

```
<div class="page-style">
```

```
<div id="content">
```

```
<a href="http://org.eclipse.ui.intro/showPage?id=page1" id="firstLink">
<font color="#376b8b" size="3"><b>Page 1</b></font>
```

```
</a>
```

```
<br/>
```

```
<a href="http://org.eclipse.ui.intro/showPage?id=page2">
<font color="#376b8b" size="3"><b>Page 2</b></font>
```

```
</a>
```

```
</div>
```

```
</div>
```

```
</body>
```

```
</html>
```

代码中标记说明如下：

在这些XHTML文件中使用超链接标签时，不能直接引用目标页面，而必须通过欢迎功能所提供的转向URL `http://org.eclipse.ui.intro/showPage?id=+页面的id`，才能正常显示目标页面。这里的id就是在introContent.xml中注册的id。

现在启动RCP程序，出现了如图20-36所示的欢迎页面。

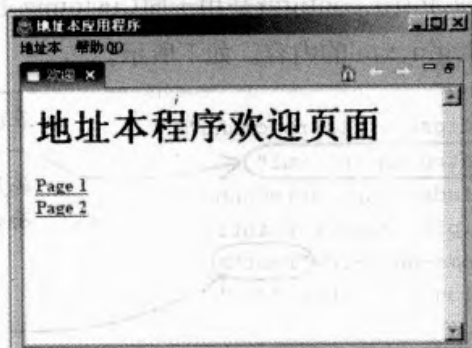
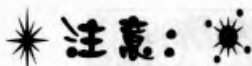


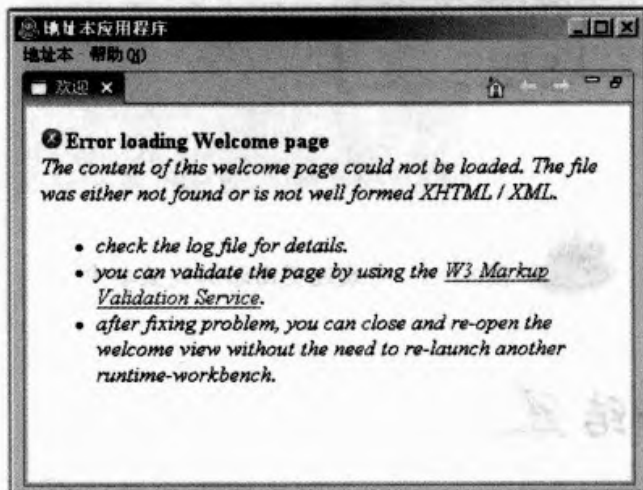
图20-36 简单的XHTML欢迎页面

这只是朴素的欢迎页面，读者可以自行开发更为绚丽多彩的页面供程序使用。



注意:

在修改XHTML页面时，有时会出现以下的错误页面。



这通常是由于XHTML文件的格式错误导致的，与普通的HTML文件不同，XHTML是XML的一种，因此对标签的关闭记号等要求都比HTML要严格得多，如HTML中的换行标签
在XHTML中一定要写成
，否则就会导致上面的错误，另外，文件编码也可能导致文件显示错误，如果要使用中文等，必须保证文件的编码为UTF-8格式。

默认情况下，欢迎页面只会在程序启动时显示一次。为了方便用户随时能够看到欢迎页面，需要在菜单中加入一个打开欢迎页面的菜单项，仍然需要修改RCP插件的操作配置类。代码如下所示（代码见光盘：\book.ch20.com.plugindev.rcp.ApplicationActionBarAdvisor）。

```
protected void makeActions(IWorkbenchWindow window) {
    .....
    welcomeAction = ActionFactory.INTRO.create(window);
    register(welcomeAction);
}

protected void fillMenuBar(IMenuManager menuBar) {
    try {
        .....
        menuManager.appendToGroup("helpContentGroup", welcomeAction); //$NON-NLS-1$
        .....
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

将系统操作“欢迎页面”加入菜单

现在, 用户通过帮助菜单中的选项就可以方便地打开欢迎页面了, 如图20-37所示。

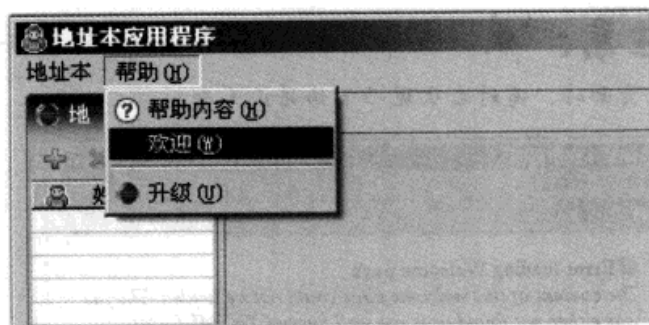


图20-37 将欢迎页面添加到菜单中

20.5 本章小结

本章从RCP程序的结构开始, 介绍了包括如何创建RCP程序, 如何将现有的插件改造成RCP程序等一系列内容。本章详细讲述了Eclipse平台下应用程序的启动过程, 包括如何创建帮助文档, 如何使RCP程序能够自动升级等话题, 不仅对RCP开发者有用, 在普通的插件开发工作中, 也是可以参考的。

学习了新的便捷方法, 真是太棒了!



第21章 Draw2d

Draw2d是一套基于SWT的图形系统，系统中提供了灵活的图形绘制功能，可以被用来创建各种自定义几何图形、复杂组合图形和不同样式的连线。相比SWT，Draw2d是一套轻量级的图形系统，使用Draw2d绘制的图形控件能够在占用较少的系统资源的情况下实现与SWT控件相同的功能。本章将要讲解Draw2d的内部结构和实现原理，以及利用Draw2d绘制各种图形元素的方法。

本章内容包括：

- ★Draw2d的组成结构和实现原理。
- ★Draw2d Figure简介。
- ★Draw2d中的坐标系统。
- ★布局管理策略。
- ★连线和路由。
- ★Draw2d中的其他组件。

拉开崭新的学习帷幕

进入第21章



21.1 理解Draw2d

Draw2d被称为轻量级（Lightweight）的图形系统，它在Eclipse平台中以独立插件的形式提供。Draw2d以Figure为基本图形元素显示图形。使用Draw2d不但可以在单个Figure内绘制任何形状的简单几何图形，也可以通过组合多个Figure构成复杂的图形。

在构建图形化定义工具的过程中，不但可以利用Draw2d绘制各种流程定义工具界面，而且可以绘制出像表单、报表编辑器这种复杂的图形编辑界面。如图21-1所示为一个使用Draw2d绘制的数据仓库ETL流程定义工具界面样式。

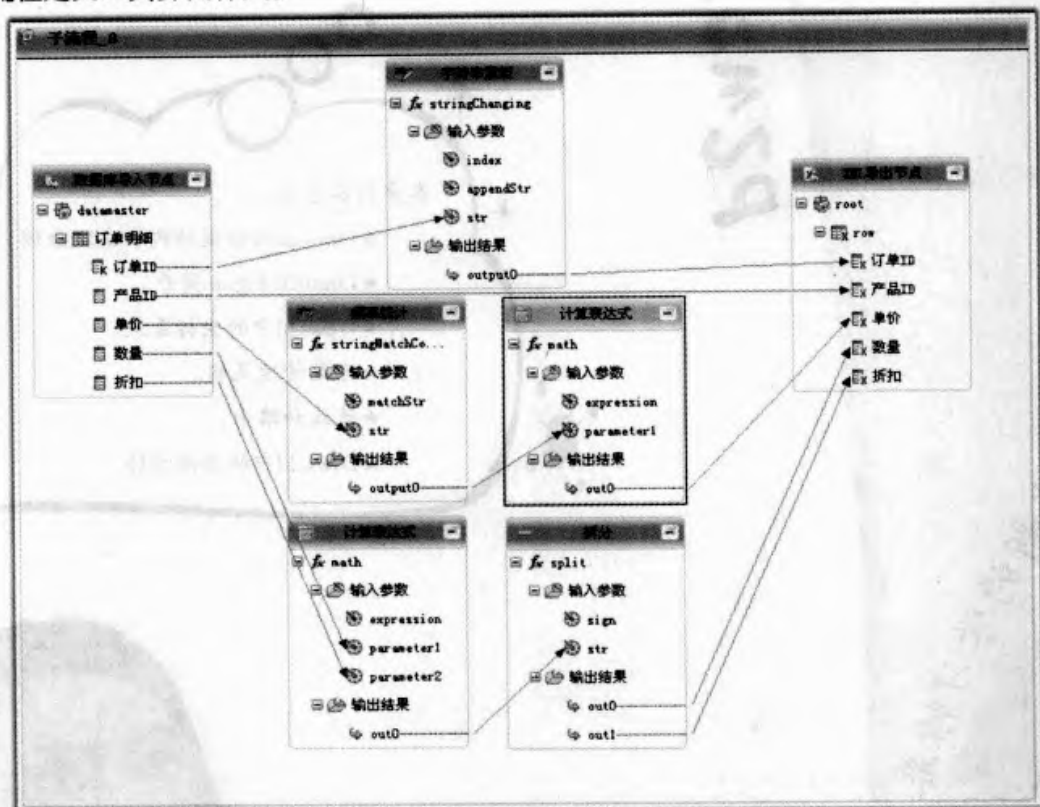


图21-1 使用Draw2d绘制的ETL流程定义工具界面

本章首先介绍Draw2d的系统架构、运行原理和各个组成部分，而后通过一个实例来了解使用Draw2d构建图形的完整过程。

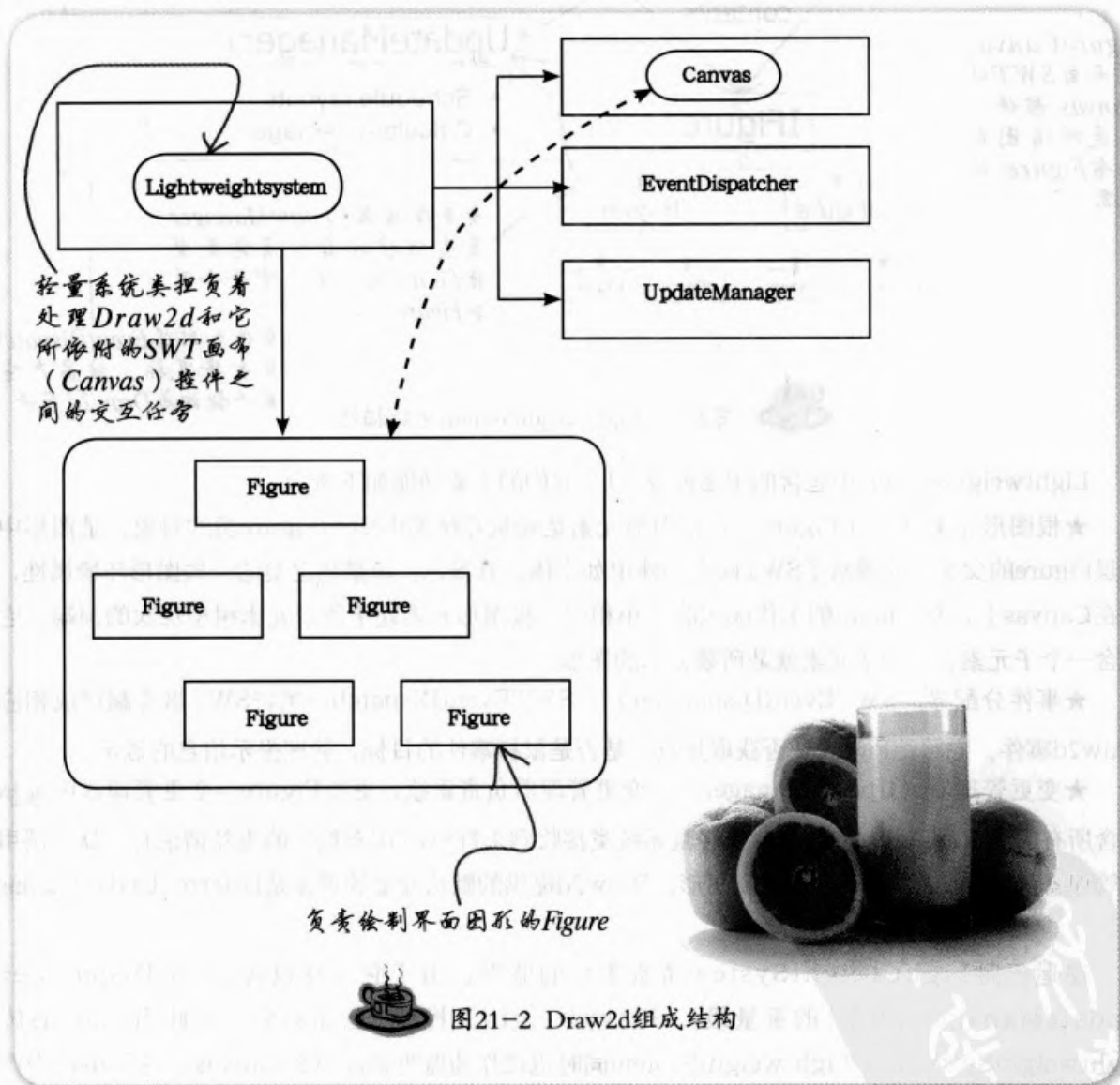
21.1.1 Draw2d系统组成

Draw2d是一套相对独立的图形系统，它以JAR包的形式提供。在使用过程中，通常情况下是和GEF一起被用来构建MVC架构系统的，但也可以在Eclipse平台中单独使用，甚至可以在Eclipse平台之外使用。Draw2d中主要包含以下几个组成部分。

★轻量系统类LightweightSystem

- ★画布Canvas
- ★图形元素Figure
- ★事件分配器EventDispatcher
- ★更新管理器UpdateManager

图21-2中描绘了Draw2d中主要组成部分之间的关系。从图中可以看出轻量系统类处于Draw2d的核心，它不但将Draw2d的其他组成联系在了一起，而且担负着处理Draw2d和它所依附的SWT画布控件之间交互的任务。



21.1.2 LightweightSystem简介

轻量系统类是Draw2d的控制中心，它把Draw2d中的其他组成部分联系在了一起。图21-3显示了LightweightSystem在整个Draw2d图形系统所起的作用和Draw2d中其他组成部分与LightweightSystem之间的关系。

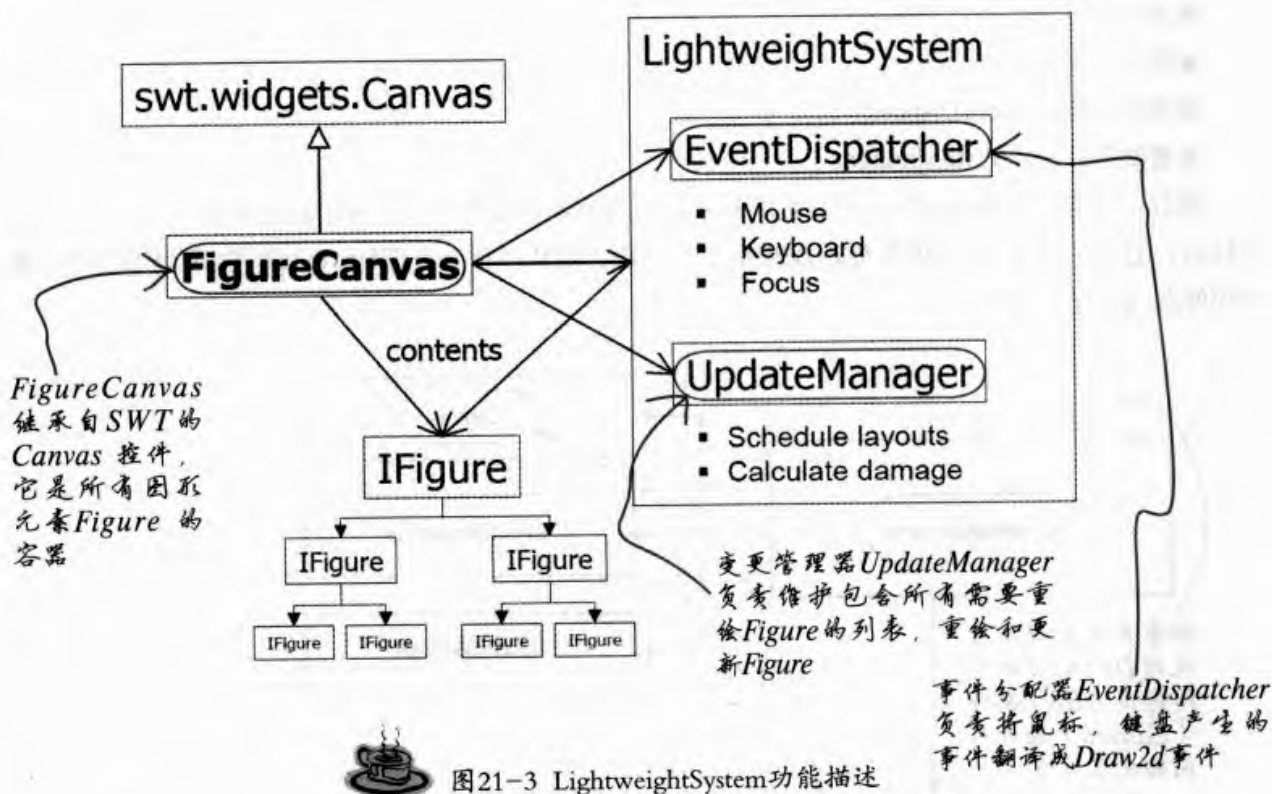


图21-3 LightweightSystem功能描述

LightweightSystem中包含的主要部分，以及它们的主要功能如下所示。

★根图形元素（RootFigure）：根图形元素是轻量系统类中RootFigure类的对象，是图形中最顶层Figure的父亲。它继承了SWT画布控件中如字体、背景色、前景色之类的一些图形环境属性，覆盖在Canvas上，与Canvas的工作区域的大小相同。根图形元素处于图形元素树型层次的顶端，它只包含一个子元素，这个子元素就是所要显示的图形。

★事件分配器（SWTEventDispatcher）：SWTEventDispatcher类将SWT事件翻译成相应的Draw2d事件。它跟踪Figure是否获取焦点，是否是鼠标事件的目标，管理提示信息的显示。

★变更管理器（UpdateManager）：变更管理器负责重绘，更新Figure。变更管理器中维护着包含所有需要重绘Figure的列表。当轻量系统类接收到来自SWT画布控件的更新请求时，就会调用变更管理器performUpdate()方法更新图形。Draw2d提供的默认变更管理器是DeferredUpdateManager类。

在运行时LightweightSystem负责事件的监听，由于它本身包含EventDispatcher，UpdateManager和SWT的重量级（Heavyweight）控件—画布对象，因此当Canvas传入LightweightSystem时，LightweightSystem同时也被作为监听器注册到Canvas。当Canvas中产生LightweightSystem所关心的事件，LightweightSystem中的某些方法就会被调用，而后这些SWT事件会被转换为Draw2d事件，通过EventDispatcher分配到当前选中的Figure图形元素中。变更管理器负责管理图形的重绘和更新当Canvas需要重绘任务。LightweightSystem会从来自Canvas的重绘事件中获取无效（dirty）矩形区域，同时请求变更管理器更新该无效矩形区域。当Draw2d中的某个图形元素无效时，Draw2d也会用变更管理器更新无效的图形元素。Draw2d中完整的事件处理流程和界面图形更新过程如图21-4所示。

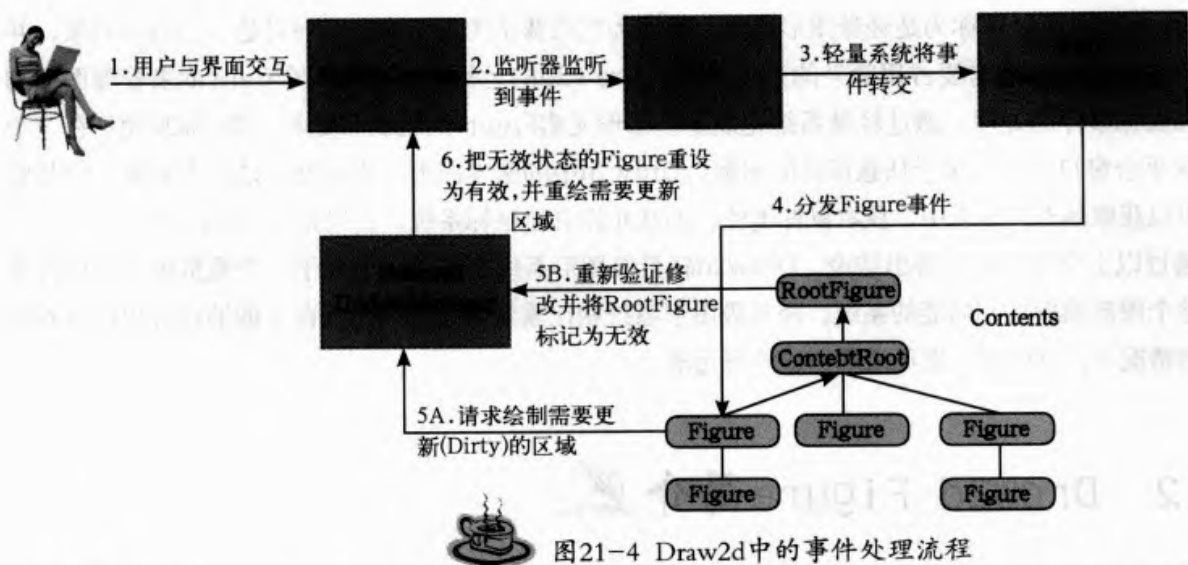


图21-4 Draw2d中的事件处理流程

注意：

当处于最底层的Figure变为无效状态后，它的所有上层Figure也将被设为无效状态。

图21-5描述了一个Draw2d的简单HelloWorld程序实现(代码见工程HelloDraw2d, sample, HelloWorld.java)。程序中通过SWT Shell显示一个Draw2d的Label控件，Label中显示“Hello World”字符串。Label是Draw2d中提供的能够显示字符串和图片的Figure图形元素。

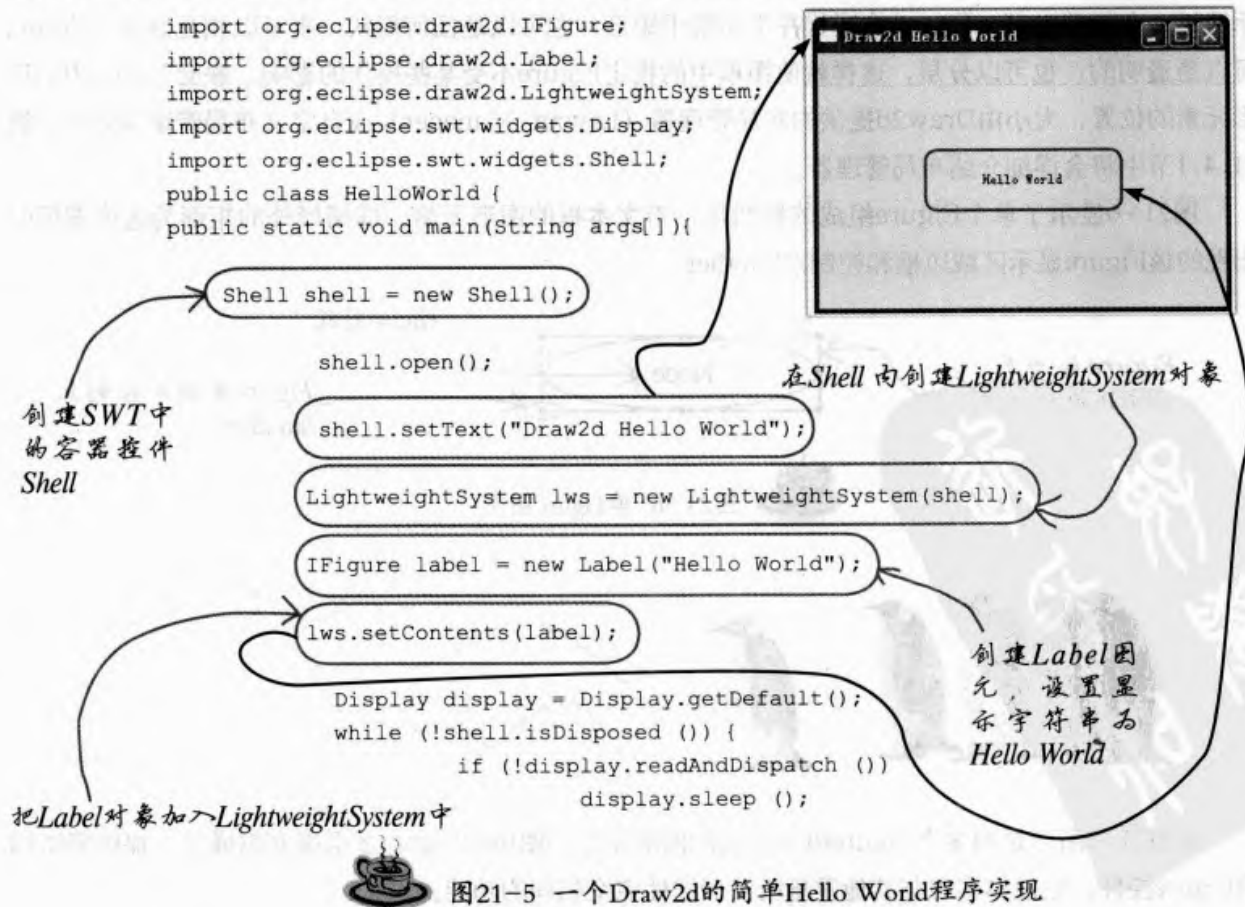


图21-5 一个Draw2d的简单Hello World程序实现

Draw2d之所以被称为是轻量图形系统，是因为它的基本图形元素Figure只是一个Java对象，并不像其他重量级控件需要占用额外的系统资源。Draw2d的运行仅依赖于一个Canvas来管理图形的显示和鼠标事件的处理，通过轻量系统完成基本图形元素Figure的绘制和更新。Draw2d允许在一个Eclipse平台窗口中显示多个任意形状的图形。Draw2d中的基本图形元素虽然只是一个对象，但是它不但可以获取焦点，被选中，获取鼠标事件，而且允许自建坐标系统，设置光标（Cursor）。

通过以上介绍，可以得出结论，Draw2d轻量级图形系统指的是仅依赖于一个重量级系统控件来实现整个图形编辑控制功能的系统。轻量级图形系统相比重量级系统的优点在于能够在占用较少系统资源的情况下，更快速、更灵活地构建图形元素。

21.2 Draw2d Figure简介

Figure是Draw2d中负责图形显示的类，它实现了IFigure接口，是Draw2d中的基本图元单位。Figure图形的实现只对应一个Figure对象，所以被称为是轻量级的图形元素。Figure并不具有图形绘制能力，它在视图中依赖Graphics显示。通常情况下，在单独的图形中只存在一个根Figure（root Figure），图形中的所有其他Figure都是其子Figure。一个独立的图形元素可以只包含一个Figure也可以由多个Figure构成。

Figure在Draw2d中的地位基本相当于其他图形界面中的控件。每一个Figure都有一个矩形的显示区域。在矩形内部，Figure的形状并不局限于矩形，它可以是任何形状，也可以相互嵌套。Figure可以是透明的，也可以分层，这样就使图形中的指定Figure不受某些操作的影响。嵌套Figure中的图形元素的位置，大小由Draw2d提供的布局管理器（Layout Manager）或自定义布局管理器管理。第21.4.1节中将会详细介绍布局管理器。

图21-6显示了单个Figure组成的椭圆形、带文本框的图形元素。椭圆形外的矩形为选中图形时出现的该Figure显示区域边框和控制点handler。

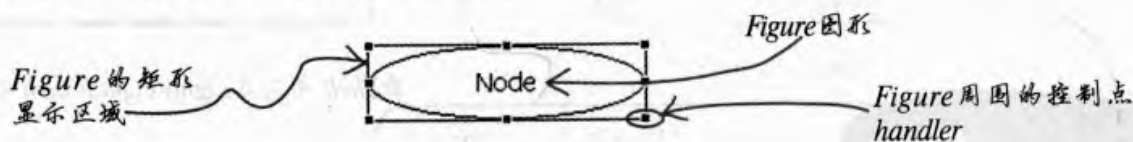


图21-6 单Figure图形



如图21-7所示是由多个Figure组合而成的图形元素。图中由Figure多层嵌套组成了一棵树形结构的Figure控件，它能够实现与其他重量级图形控件完全相同的功能。

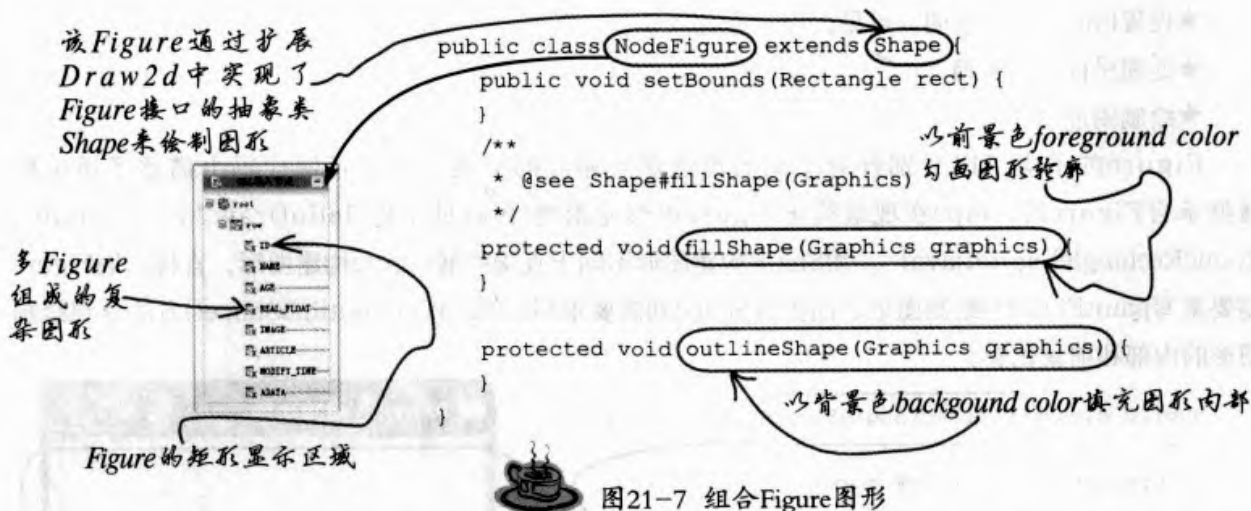


图21-7 组合Figure图形

Figure是图形显示的基础，其子类包括Clickable, FlowFigure, ImageFigure, Label, LabeledContainer, Layer, LightweightSystem, RootFigure, Panel, ScrollBar, ScrollPane, Shape, Thumbnail, Viewport等。这些子类分别对Figure做了扩展，用以实现不同的功能。图21-8显示了Figure与它的这些子类之间的关系，这些Figure的具体功能和用法在Draw2d的相关帮助文档中有详细介绍，这里就不一一列举了。

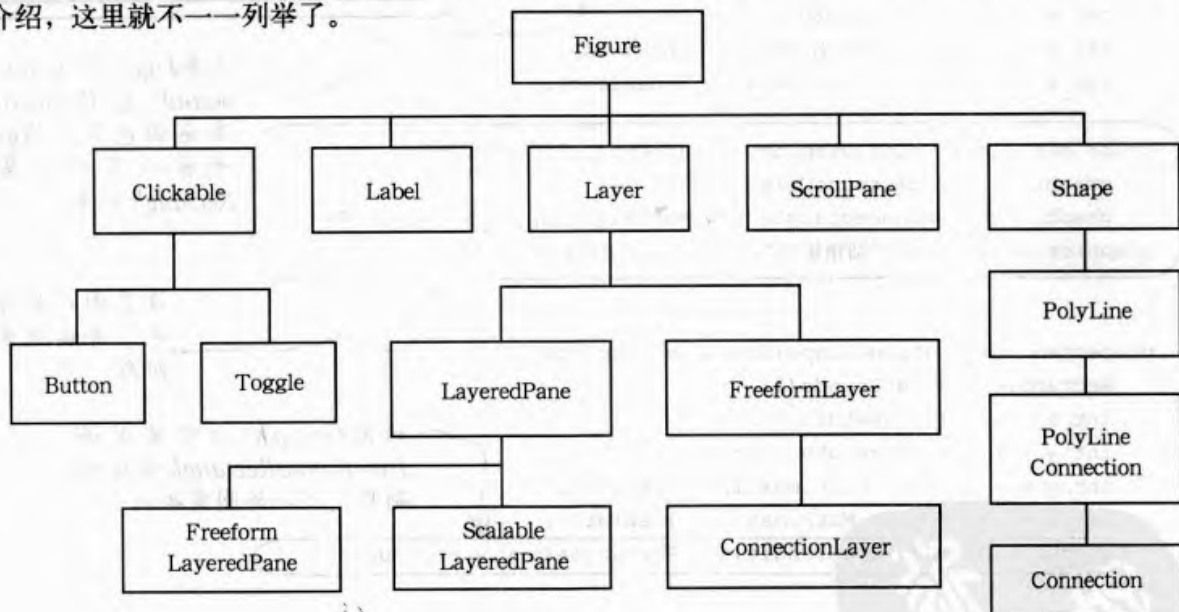


图21-8 Draw2d中Figure的主要子类

图21-8中的这些类只是Figure所有子类中最常用的一部分。与SWT的Control相似，Figure中也包含了大量的方法更改其属性。这些方法按功能大致可分为以下几类。

- ★在Figure上注册或取消注册监听器 (Listener)，以监听产生在Figure上的鼠标事件。
- ★处理如移动Figure，改变Figure大小之类的图形结构事件。
- ★当鼠标进入时，指定鼠标形状。
- ★处理多个Figure嵌套图形元素中的子Figure位置改变、获取、添加、删除、获取父Figure等操作。
- ★设置、获取焦点。

★设置Figure是否透明、可见。

★处理坐标系统转换。

★绘制图形。

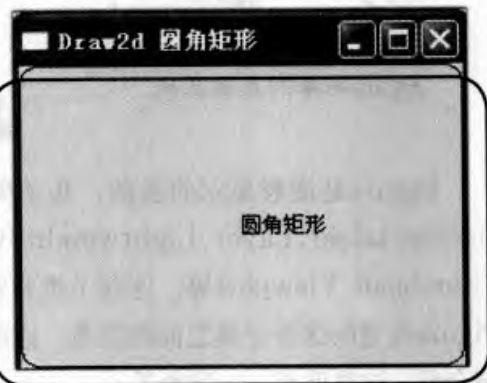
Figure的子类可以分别针对不同的用途加入相应的扩展，下面示例代码中描述了通过扩展继承自Figure的Shape实现最简单Figure图形元素类(代码见工程HelloDraw2d, sample. RouncRectangleFigure.java)。用Shape构建图形不同于直接扩展Figure构建图形，直接扩展Figure需要重写paintFigure()绘制图形，而扩展Shape则需要重写fillShape()和outlineShape()方法分别绘制图形的内部和前景轮廓。

```
public class RouncRectangleFigure extends Shape {
    /**
     * Creates a RectangleFigure.
     */
    public RouncRectangleFigure() { }
    protected void fillShape(Graphics graphics) {

        Rectangle r = getBounds();

        int x = r.x + lineWidth / 2;
        int y = r.y + lineWidth / 2;
        int w = r.width - Math.max(1, lineWidth);
        int h = r.height - Math.max(1, lineWidth);

        Color clr = new Color(Display.getCurrent(), 255, 255, 255);
        graphics.setBackgroundColor(clr);
        graphics.fillRoundRectangle(new Rectangle(x, y, w, h), 20, 20);
        graphics.drawString("圆角矩形", r.getCenter());
    }
    protected void outlineShape(Graphics graphics) {
        Rectangle r = getBounds();
        int x = r.x + lineWidth / 2;
        int y = r.y + lineWidth / 2;
        int w = r.width - Math.max(1, lineWidth);
        int h = r.height - Math.max(1, lineWidth);
        graphics.drawRoundRectangle(new Rectangle(x, y, w, h), 20, 20);
    }
}
```



获取Figure的绘图区域
bound, getBounds()方
法返回包含有Figure
坐标和宽高信息的
Rectangle对象

设置图形的背景
色，并填充圆角
矩形

利用Graphics中提供的
drawRoundRectangle方法绘
制指定大小的圆角矩形

※ 注意: ※

这里不要在fillShape方法中绘制图形轮廓或是在outlineShape方法中填充图形。因为在Shape的实现中，这两个方法会在paintFigure方法中按顺序调用，如果在程序中用Shape中提供的setLineStyle或setLineWidth方法设置了边框线条的样式，那么fillShape中绘制的边框样式将不会随之改变。



21.3 坐标系统

Draw2d中提供了灵活的坐标系统，方便了图形的定位与显示。坐标系统的选择并不影响Figure

绘制自身结构，它只决定Draw2d的Graphics绘制其子图形元素时的矩形绘图区域 — Bounds。Draw2d中提供的坐标系有绝对坐标和相对坐标两种。

1. 绝对坐标 (Absolute Coordinate)

绝对坐标系统中Figure和所属的所有子Figure以同一坐标定位。当绝对坐标系统中的Figure位置移动一定距离后，其子Figure坐标也要改变相同的偏移量。

2. 相对坐标 (Relative Coordinate)

相对坐标指的是图形元素相对于其父容器的坐标系。它使处于相对坐标系中的图元只需要关心其相对父容器的位置。当Figure位置改变时，其子Figure坐标大小不变。

在表21-1中，通过对完成不同功能的过程的对比描述了两种坐标系所适用的不同场合及其使用限制。

表21-1 绝对坐标和相对坐标对比

任务	绝对坐标	相对坐标
移动Figure时	在移动Figure时，需要重新计算Figure本身和所属的子Figure位置。在某些情况下这将很耗时	只需要计算被移动Figure本身位置、大小，其子Figure位置、大小自动计算确定
确定重绘区域	不需要对坐标系做调整即可确定重绘区域	利用简单算法即可调整坐标系，得到重绘区域
获取Figure在Canvas上的位置	如果Figure上层Figure均采用绝对坐标，可以利用FigureListener获取Figure在Canvas上的位置	必须利用FigureListener和CoordinateListener，并调用Figure中的translateToAbsolute方法获取位置坐标
当子Figure大小、位置计算完后，计算父Figure大小和位置	子Figure位置、大小确定之后，父Figure可以很容易地计算出自己的大小、位置	由于Figure位置、大小的改变都会引起其子Figure位置、大小的改变，所以用相对坐标完成此任务相对困难

默认情况下，Draw2d中采用绝对坐标系。通过重写 useLocalCoordinate()方法，Figure可以改变其所包含的子Figure所在的坐标系。

21.4 布局管理

在实现Draw2d的嵌套图形元素时，一个Figure往往会嵌套多个子Figure，这些子Figure的位置和大小可以由布局管理器来管理。布局管理器提供了一些布局算法来计算各Figure的位置和大小，同时也负责管理附加在各个Figure之上的各种约束 (Constraint) 信息。

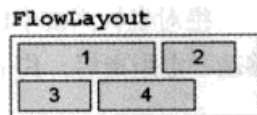
21.4.1 布局管理器

Draw2d中的布局管理交由布局管理器负责。在管理布局时，布局管理器首先逐个计算在其管理之中的各个子Figure的最佳大小值，然后用布局算法计算出子Figure的实际大小和位置信息并传递回

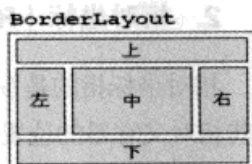
子Figure。

所有的Draw2d布局管理器都继承自抽象类AbstractLayout。布局方式不同，对Figure的约束限制也会不同。如XYLayout布局管理要求它所管理的Figure具有Rectangle类型的约束，而DelegatingLayout布局则需要Figure提供实现Locator接口的定位器。Draw2d中提供了多种布局管理器，其中包括如下所示内容。

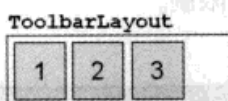
★FlowLayout：以行的方式排列所有其管理之中的Figure；提供调整Figure的对齐方式和间距的方法。



★BorderLayout：以边界方式管理布局。这种方式将区域划分为上、下、左、右、中5部分管理Figure之间的位置、间距和大小信息。



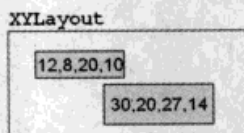
★ToolBarLayout：按照工具栏的风格布局Figure，多用于布局工具栏控件。



★StackLayout：用堆栈的风格管理Figure布局。Figure的摆放顺序取决于Figure被加入的次序，首先加入的Figure位于布局的最底端。

★DelegatingLayout：这种布局方式需要Figure提供实现Locator接口的定位器，依赖定位器完成Figure的布局管理。

★XYLayout：以坐标方式管理Figure的布局。这种方式允许设置其管理中Figure的坐标和矩形区域大小。



★ScrollPaneLayout：管理滚动条和包含滚动板（ScrollPane）的视图（Viewport）的布局方式。

★ViewprotLayout：管理视图的可视区域和滚动位置状态。

21.4.2 布局更新

Draw2d中判断Figure是否需要更新布局要看Figure是否有效（Validate），所有Figure在创建时状态都是无效的。在创建后的某个时刻，通过调用validate方法Figure会被标识为有效，同时Figure也会被重新布局。如果在图形编辑的过程中某个Figure发生更新就会重新被标识为无效。

图形中各个Figure图形元素之间的结构可以用一棵树来表示，树根就是Root Figure，结构如图21-9所示。在绘制过程中，按照前序，深度优先顺序进行。过程中唯一例外是边界，它是以后序绘制的。Figure的绘制顺序一般情况下是按照Z轴顺序（Z-axis-order），如图21-10所示。

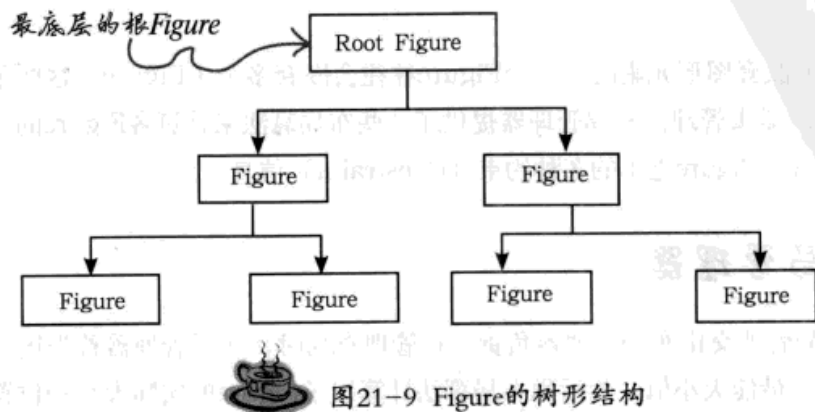


图21-9 Figure的树形结构

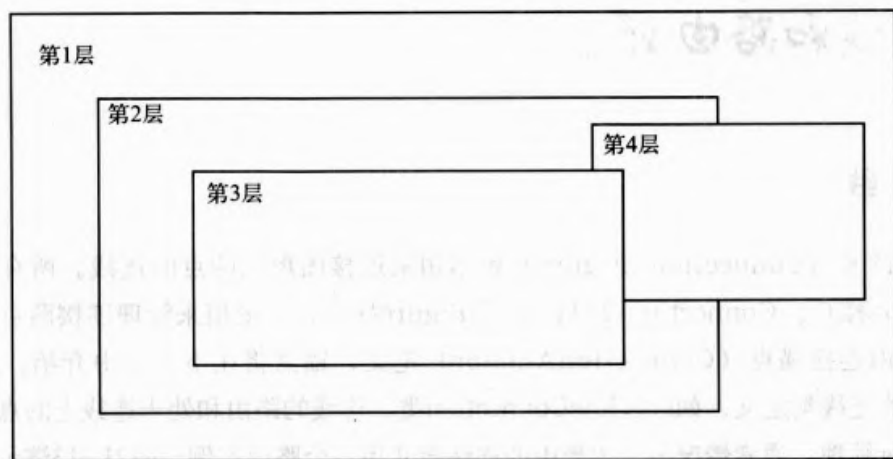


图21-10 多层Figure的Z序



按照此规则，Figure不但可以更新本身和它所包含的Figure，也可以请求更新底层的Figure，图21-11和图21-12描绘了当最底层图形元素产生更新时，整个Figure树标记有效状态和更新的顺序。图中灰色的Figure代表此次更新影响到的Figure。

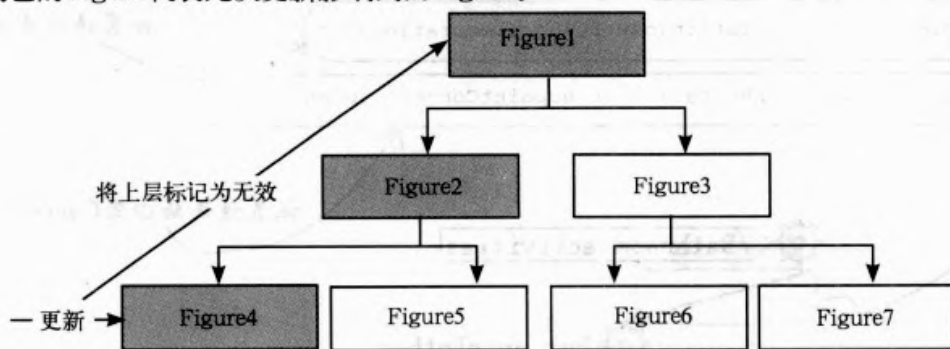


图21-11 请求更新过程

当Figure的最佳大小和布局方式发生改变时，其本身会调用revalidate()方法。revalidate()方法会将该Figure本身和直到根Figure的所有底层相关Figure标识为无效，同时加入到更新管理器（update manager）中；然后，更新管理器按顺序调用Figure的validate()方法重新布局。

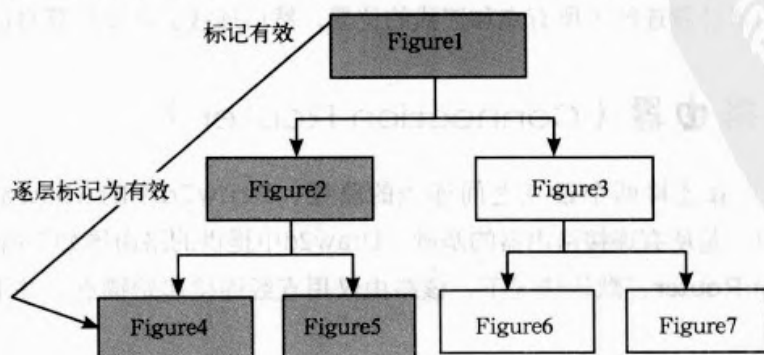


图21-12 更新过程

21.5 连线和路由

21.5.1 连线

Figure连线图元 (Connection Figure) 表示用来连接图形上两点的连线。所有连线图元都实现自Connection接口。Connection接口扩展了IFigure接口, 主要用来管理连接两点间的线条。连线起点与终点由连接锚点 (ConnectionAnchor) 定义, 锚点将在下一节中介绍。连线由实现Connection接口的连线类定义, 如PolylineConnection类。连线的路由和处于连线上的点的位置则由ConnectionRouter管理。通常情况下, 图形中的连线会共用一个路由实例。图21-13演示了带有连接线的图形样式和连接线图形的创建方法。

```
public class ConnectionPart extends AbstractConnectionEditPart {  
    protected IFigure createFigure() {
```

```
        PolylineConnection conn = new PolylineConnection();
```

创建PolylineConnection连接

```
        conn.setTargetDecoration(new PolygonDecoration());
```

设置连线的标识

```
        conn.setConnectionRouter(new BendpointConnectionRouter());
```

设置连线路由器Connection Router

```
        return conn;
```

```
    }
```

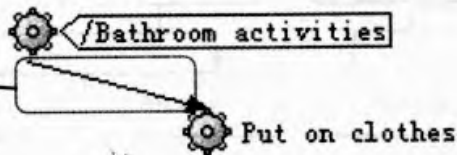


图21-13 带有连接线的Draw2d图形

连线的创建和添加方式与其他Figure相同, 连线一般显示在其他以层次显示的图形元素之上, 与其他Figure不同的是连线不需要为其设置位置与大小。在显示连线时需要为Connection提供起点和终点锚点, 并让连接路由器计算连线上所有点和连线的位置, 然后连线会自动计算自己的显示区域。

21.5.2 连线路由器 (Connection Router)

连线路由器负责计算连接两个锚点之间连线的路径。Draw2d中的AbstractRouter实现了ConnectionRouter接口, 是所有连接路由器的基础。Draw2d中提供的路由器如下所示。

★NullConnectionRouter: 默认情况下, 该路由仅用直线连接起始锚点, 并不提供其他路由计算。

★AutomaticRouter: 此路由为防止连线重叠路由器的基类。继承此类的路由器会将起点和终点相同的连线散开显示。

★**BendpointConnectionRouter**：该路由器允许用户以任意拖动连线中某部分的方式添加连线转折点，路由算法能够自动计算连线上各点的位置。连线样式如图21-14所示。

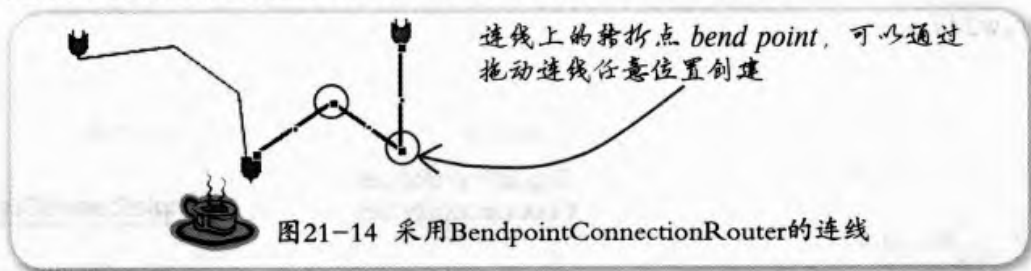


图21-14 采用BendpointConnectionRouter的连线

★**ManhattanConnectionRouter**：曼哈顿线方式的连接路由原则上只以垂直和水平线来组成连接连线起点和终点的线条，其中也提供了回避线条重叠的功能。连线样式如图21-15所示。

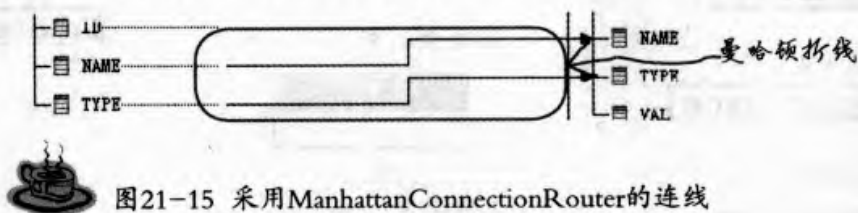


图21-15 采用ManhattanConnectionRouter的连线

21.6 Draw2d中的其他组件

本节介绍一些Draw2d中会被经常用到的其他组件，要想创建出需要的Draw2d图形来，必须灵活掌握这些组件的用法。这些常用组件包括边界、层次、定位器、连接锚点。

21.6.1 边界

Draw2d中创建的Figure图形元素往往需要添加一个边界来装饰。边界（Border）一般显示在Figure的显示区域内，可以通过设置边界偏移量（Inset）调整边界内的子Figure与边界的相对位置。在Draw2d中提供了诸如标题栏边界、框架边界、线条边界等边界类。边界类实现了Border接口，提供不同的显示效果。常用的边界类包括下面几种。

- ★**分组框边界（GroupBoxBorder）**：创建类似于Windows系统中分组框风格的带标签的边界。
- ★**标题栏边界（TitleBarBorder）**：创建类似窗口标题栏边框风格的边界。
- ★**复合式边界（CompoundBorder）**：由两个边界组成的边界。
- ★**框架边界（FrameBorder）**：类似于标题栏边界，可以用来创建带标题栏的Figure。
- ★**焦点边界（FocusBorder）**：矩形框式边界，表示Figure获取焦点。
- ★**线条边界（LineBorder）**：用指定宽度的线条勾画Figure轮廓。
- ★**边框边界（MarginBorder）**：创建带边缘空白的边界衬托Figure。
- ★**预配置型边界（SchemeBorder）**：通过添加包含预设置好边界属性信息的Scheme类来创建的边界。这种边界可以显示出阴影、高亮显示等效果。
- ★**下沉式边界（SimpleLoweredBorder）**：以下沉凹陷风格显示的边界。

★上浮式边界 (SimpleRaisedBorder)：以上浮凸起风格显示的边界。

图21-16显示了以上描述的各种Draw2d边界的各种显示样式，读者可以参照这些样式来设计自己的Draw2d界面。

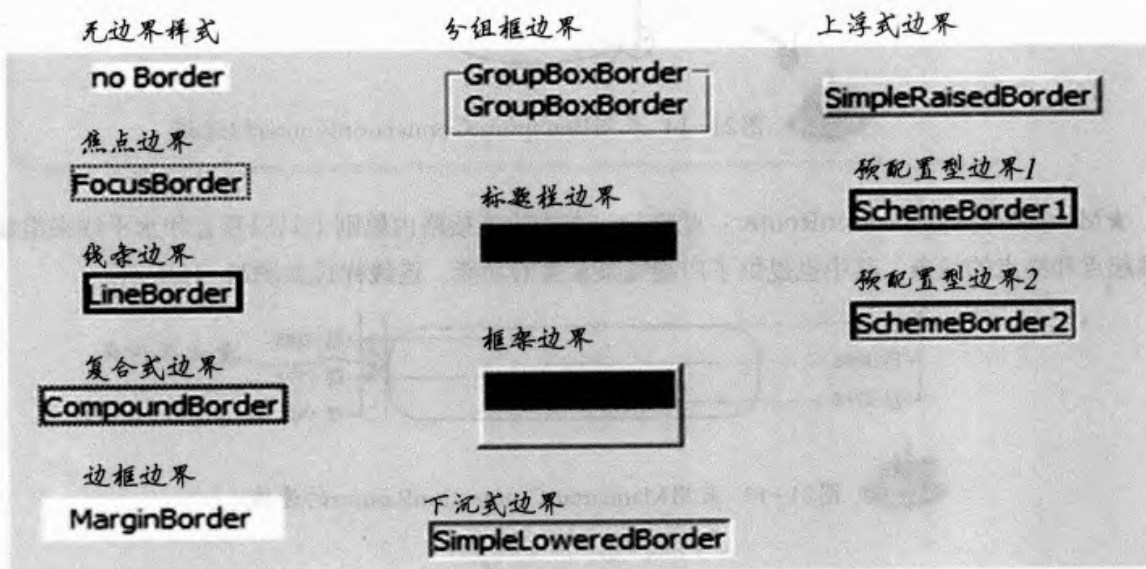


图21-16 Draw2d中各种边界 (Border) 样式

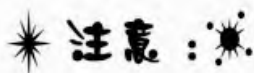
21.6.2 层次

层次 (Layer) 是一种用在层次窗口 (LayerPane) 中的透明的Figure，是用来维护层次的容器，它包含一个以字符串类型为Key，层次为Value的map。层次窗口中包含添加、删除、插入、排序其所维护层次的方法。它实现了IFigure接口中的containsPoint()和findFigureAt()方法以使单击操作能够作用到层次下面的Figure。

下面代码演示了以从桔红色到白色渐变色填充的扩展自FreeformLayer的背景层次。

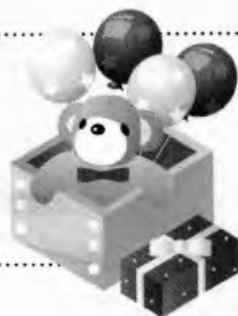
```
public class BackgroundLayer extends FreeformLayer {
    public static final String BACKGROUND_LAYER = "Background Layer";
    public BackgroundLayer() {
        setOpaque(true);
    }
    protected void paintFigure(Graphics graphics) {
        if(isOpaque()) {
            graphics.setForegroundColor(ColorConstants.white);
            graphics.setBackgroundColor(ColorConstants.orange);
            graphics.fillGradient(getBounds(), true);
        }
    }
}
```

设置白色前景色，桔红色背景色，垂直渐变填充



注意：

*BackgroundLayer*中定义的静态字符串*BACKGROUND_LAYER*必须能够唯一标识处于同一*LayerPane*中的层次，否则将覆盖*LayerPane*中同名的*Layer*。



21.6.3 定位器

定位器 (Locator) 在Draw2d中负责标定Figure的位置。定位器实现了接口Locator，Locator中包含唯一的方法void relocate(IFigure target)。定位器可以确保在Connection位置发生改变时Figure始终与Connection连接。例如，ConnectionLocator是一种定位器，它的功能是定位连接到Connection的Figure。ConnectionLocator可以用来设置连接末端箭头位置或在连接上放置标签或解释。

Draw2d中有如下所示的定位器。

★**ArrowLocator**：用来定位连接末端如箭头之类的连线符号。实现RotatableDecoration的Figure都可以被定位。

★**BendpointLocator**：用来定位折线连线折点的锚点位置的定位器。

★**MidpointLocator**：定位处于连线中间点位置Figure的定位器。

★**ConnectionEndpointLocator**：用来定义处于连线起点和终点位置Figure的定位器。

★**RelativeLocator**：此定位器用来定位以取值在0和1之间的浮点数来表示在父Figure中左上角为0，右下角为1相对位置的子Figure的位置。

21.6.4 连接锚点

连接锚点 (Connection Anchor) 是用来表示连接线端点的标识。锚点类中包含的基本方法的功能包括维护锚点位置信息和注册监听锚点移动。Draw2d中提供了各种样式的锚点，其中常用锚点包括如下几种。

★**ChopboxAnchor**：当连接Figure的连线以Figure中心点为终点时，连线与Figure边界的交点位置就是ChopboxAnchor的位置。ChopboxAnchor锚点可以随着连线位置的改变而在Figure边界上运动。

★**LabelAnchor**：LabelAnchor是ChopboxAnchor的子类，它是为Draw2d标签Figure单独设计的。LabelAnchor锚点的位置只与标签Figure图标中心位置有关。

★**EllipseAnchor**：顾名思义，EllipseAnchor是椭圆 (Ellipse) Figure边界和连接到Figure中心点连线相交的交点位置锚点。它是ChopboxAnchor锚点类中的一个变量。

★**XYAnchor**：XYAnchor锚点用于描绘指定X，Y坐标位置的锚点。

下面就以ChopboxAnchor为例分析一下锚点的实现。

图21-17显示了ChopboxAnchor显示的位置和连接样式。

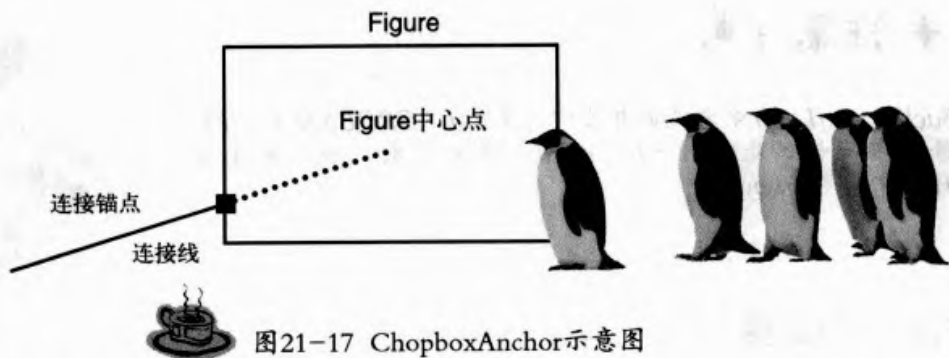


图21-17 ChopboxAnchor示意图

首先来看一下ChopboxAnchor类的结构。ChopboxAnchor类扩展自AbstractConnectionAnchor类，其内部结构如图21-18所示。

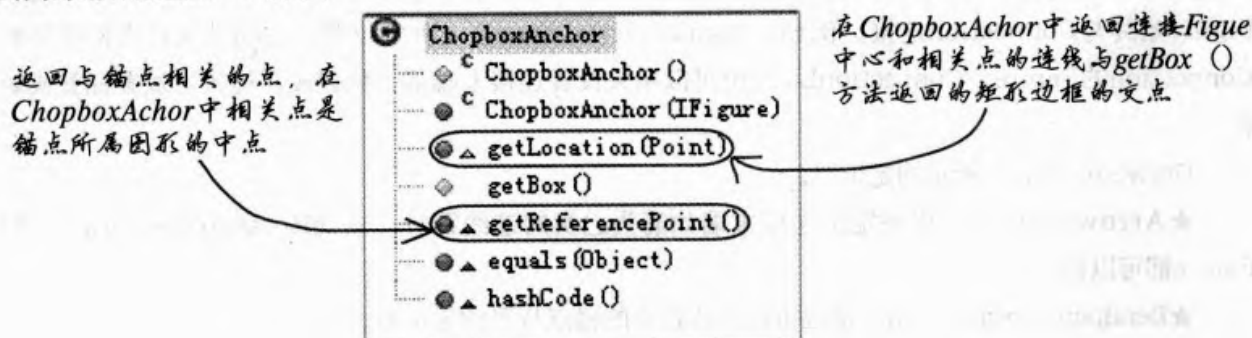


图21-18 ChopboxAnchor类

由图21-18中可以看出ChopboxAnchor类中主要方法是getLocation()，其功能是计算锚点位置。方法getLocation()的实现代码如下所示。

```
public Point getLocation(Point reference) {
    Rectangle r = Rectangle.SINGLETON;
    r.setBounds(getBox());
    r.translate(-1, -1);
    r.resize(1, 1);
    getOwner().translateToAbsolute(r);
    float centerX = r.x + 0.5f * r.width;
    float centerY = r.y + 0.5f * r.height;
```

如果getBox 返回矩形为空，或点reference与Figure中心点重叠，返回Figure中心点

```
if (r.isEmpty() || (reference.x == (int)centerX && reference.y == (int)centerY))
    return new Point((int)centerX, (int)centerY); //This avoids
```

```
float dx = reference.x - centerX;
float dy = reference.y - centerY;
float scale = 0.5f / Math.max(Math.abs(dx) / r.width, Math.abs(dy) / r.height);
dx *= scale;
dy *= scale;
centerX += dx;
centerY += dy;
return new Point(Math.round(centerX), Math.round(centerY));
}
```

否则计算矩形与连线交点并返回该点

21.7 本章小结

Draw2d中提供了丰富的图形绘制功能，为构建MVC（Model View Controller）框架中的视图（View）部分提供了支持。但是Draw2d只提供了显示模型的视图类，并没有提供与编辑相关的功能，如果利用Draw2d创建只需要图形展示的系统，单用Draw2d可以满足所有图形显示需要，但是如果提供用户图形编辑功能，那就需要图形编辑框架GEF（Graphical Editing Framework）的加盟了。下一章将介绍GEF的相关内容。



第22章

GEF介绍与实现

上一章介绍了的图形系统Draw2d, 了解了Draw2d的内部组成、运行原理以及如何用Draw2d绘制自己的图形元素。本章将讲解图形化编辑器框架——GEF, 通过实例学习如何利用GEF和Draw2d创建出Eclipse平台中的图形编辑工具。读者通过本章的学习, 将要掌握GEF架构中的概念和使用, 并能够创建出简单的GEF图形编辑工具。

本章内容包括:

- ★ GEF架构和工作原理。
- ★ 理解GEF中的MVC架构。
- ★ GEF中的基本概念。
- ★ 创建GEF图形编辑器。

拉开崭新的学习帷幕

进入第22章



22.1 GEF简介

GEF (Graphical Editor Framework) 即图形编辑器框架, 是Eclipse平台下又一个功能强大的开源工程。GEF框架为以图形展现用户自定义模型, 创建图形编辑器并通过图形操作编辑模型提供了支持。通过使用GEF可以方便地创建出多种样式的图形编辑器。在Eclipse平台中GEF能够被用在任何可以使用SWT控件的地方, 如编辑器、视图或属性页、向导页等。通常情况下GEF框架会被用来创建EditorPart图形编辑器和大纲视图。

下面先来看几个使用GEF创建的图形编辑器界面。图22-1是GEF提供的示例中的流程编辑器, 图中展示了以节点连线方式表示从起床到开车上班去的整个过程。

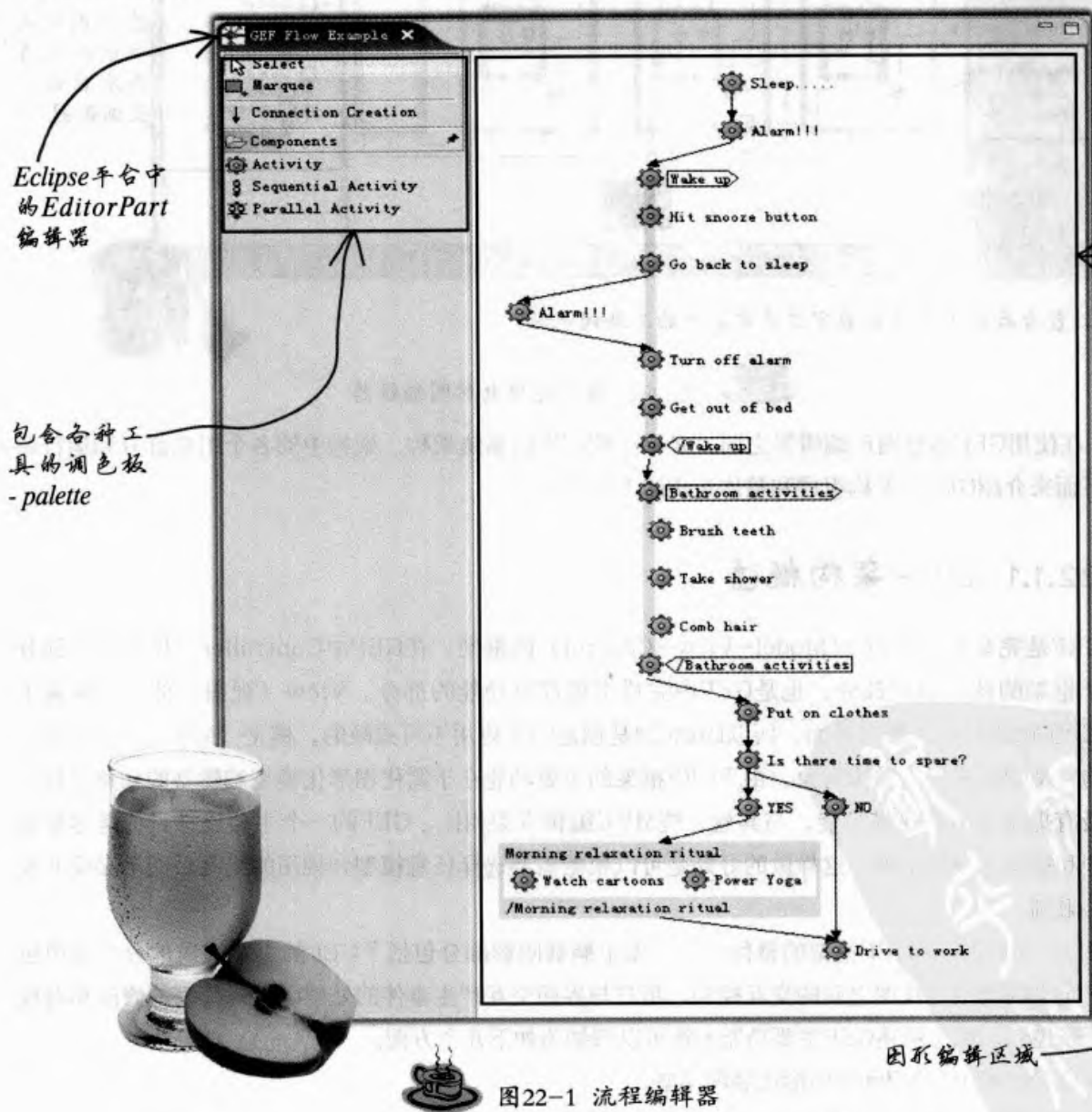


图22-2是GEF示例中的数字逻辑电路图的图形编辑器, 工具中包括了调色板、编辑器和大纲视图。

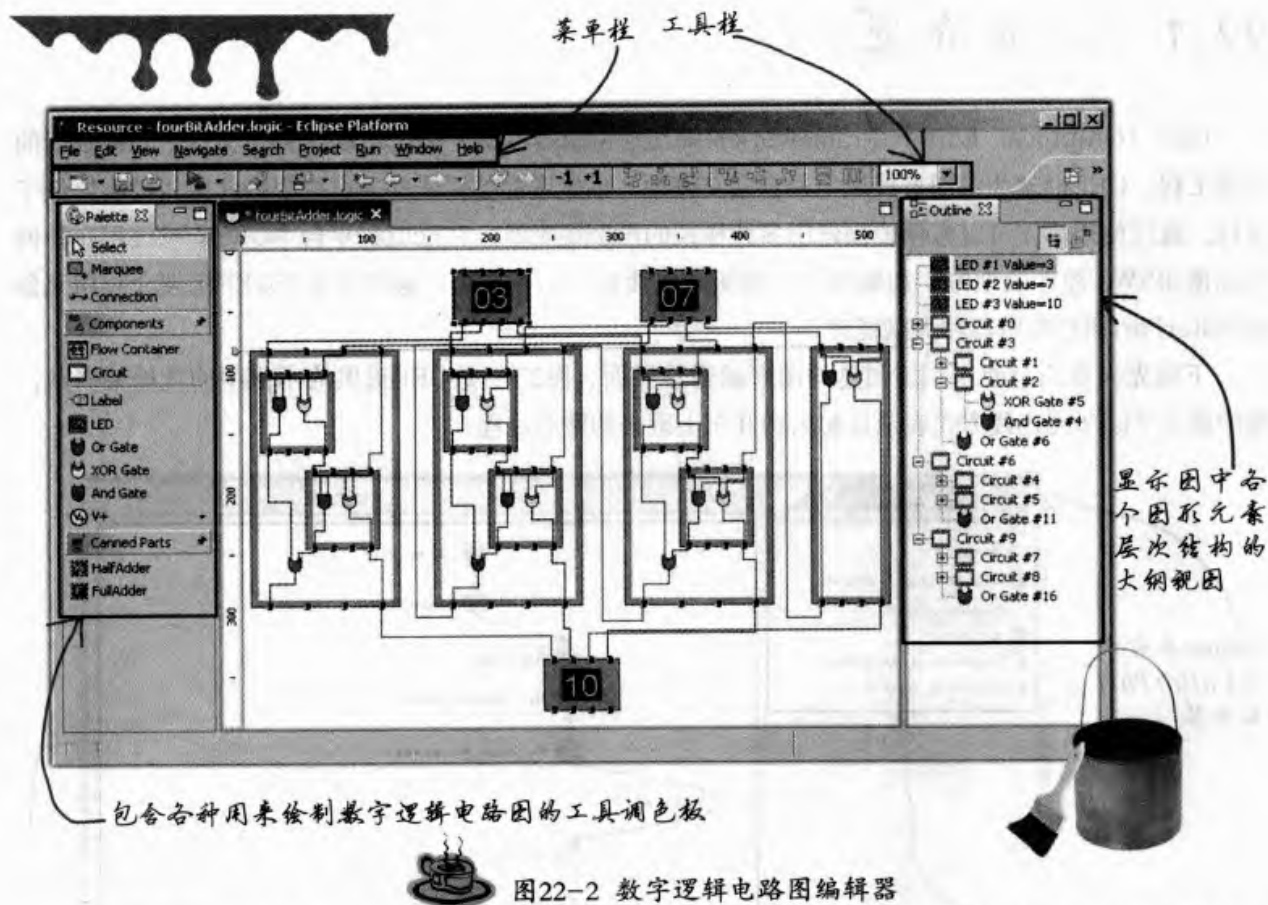


图22-2 数字逻辑电路图编辑器

在使用GEF创建图形编辑器之前，必须了解GEF的系统架构、架构中的各个组成部分和运行原理。下面来介绍GEF的架构组成和整体架构的工作原理。

22.1.1 GEF架构概述

GEF是完全基于MVC (Model-View-Control) 的框架，在GEF中Controller (控制器) 部分是整个框架的核心组成部分，也是GEF中完成主要控制功能的部分。View (视图) 部分是依赖于Draw2d实现的轻量级图形界面，所以Draw2d是创建GEF应用不可或缺的。模型 (Model) 是由用户设计的，是GEF应用的编辑对象。由于GEF框架的主要功能在于简化图形化模型编辑器的构建过程，所以没有提供太多的建模功能。与其他一些MVC编辑框架相比，GEF的一个主要设计目标是尽量减少模型和视图之间的依赖，这样做的好处是可以根据需要选择任意模型和视图的组合，而不必受开发框架的限制。

图22-3中显示了GEF框架的整体结构。图中椭圆阴影部分包括了GEF的主要组成部分，其中包含用户定义的模型与视图之间的交互控制，用户与界面交互产生事件的处理以及对模型的修改和对视图的更新控制。综上所述GEF主要功能大致可以归纳为如下几个方面。

- ★以Draw2d中的Figure图形化展现模型。
- ★支持鼠标、键盘以及Workbench间的交互操作。
- ★提供支持以上功能的通用组件。

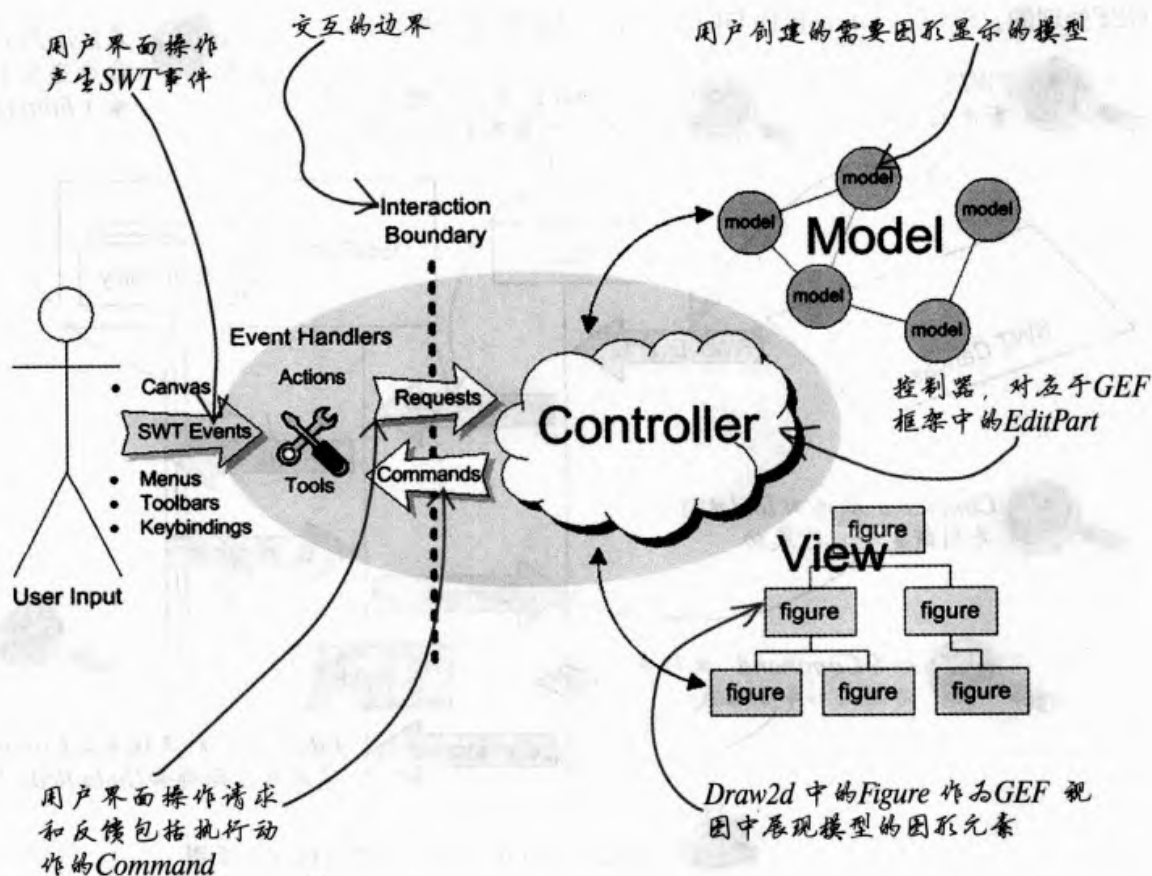


图22-3 GEF框架整体结构图

图22-4描述了GEF在图形界面中的层次关系。可以看到处于底层的是SWT的画布Canvas控件，它负责处理所有图形的显示和用户交互事件；Draw2d层则关注于绘制模型对应的图形元素的绘制和图形布局管理等功能；GEF其实并不关心图形是如何绘制的，它所关注的是如何将模型与图形元素联系在一起，并将图形界面中的修改请求转换为对模型结构和模型属性的修改。

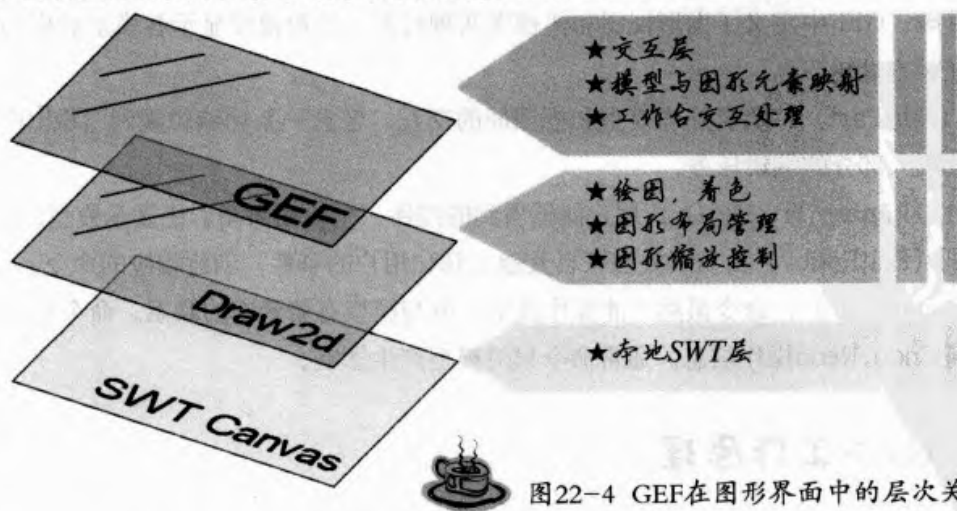


图22-4 GEF在图形界面中的层次关系

首先来看一下处于最底层的SWT Canvas控件是如何处理SWT事件 (Event) 并将事件发送给

GEF处理的。SWT Canvas控件与GEF交互过程如图22-5所示。

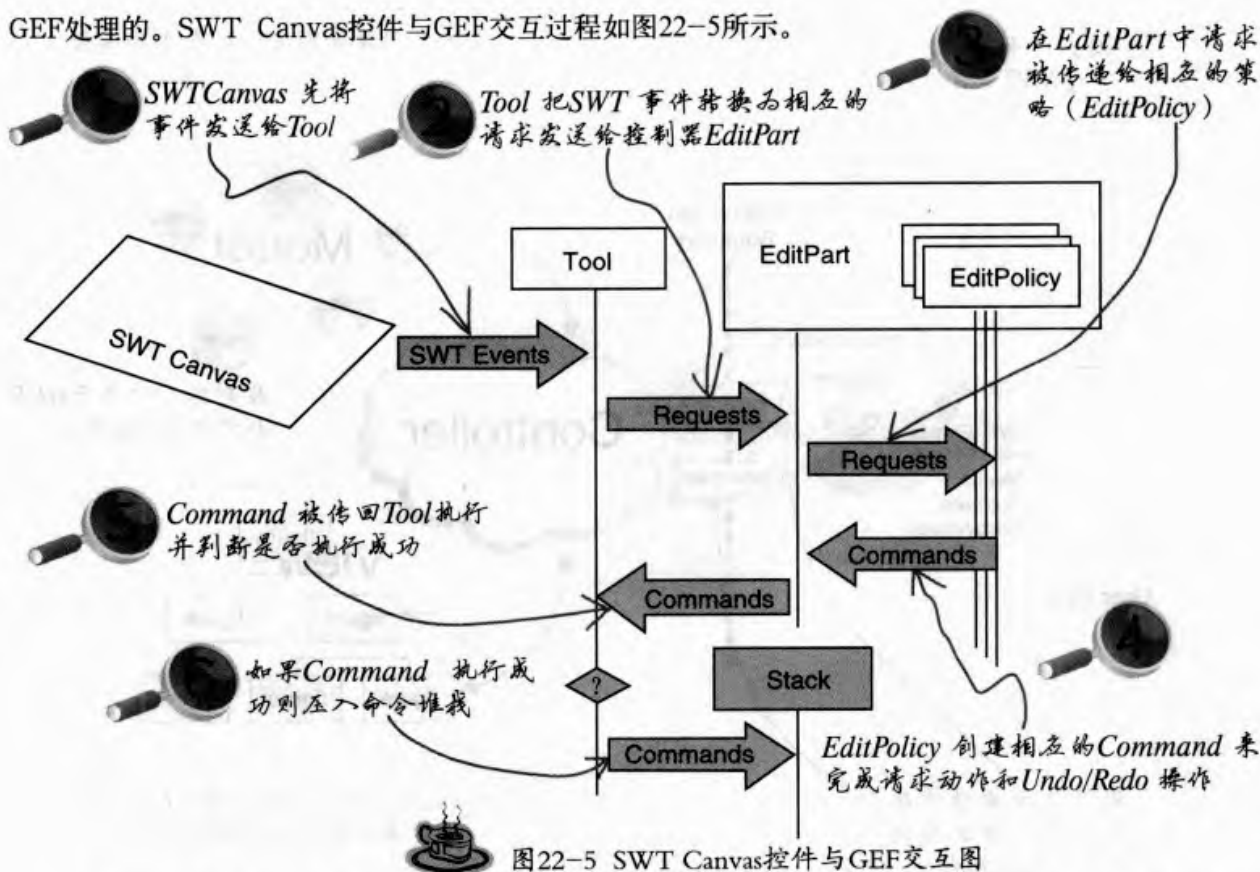


图22-5 SWT Canvas控件与GEF交互图

在用GEF设计图形界面应用的过程中，控制器是视图和模型之间交互的唯一桥梁。通常情况下请求 (Request) 处理、模型的修改和视图的更新都由控制器来完成。下面来了解一下GEF中的各个主要组成部分和它们的功能，如下所示。

★模型 (Model)：模型包含了所有需要持久化的数据，GEF中的所有模型必须要有一种交互机制来与视图更新保持一致。

★视图 (EditPartViewer)：GEF中的视图功能类似JFace中的视图，它提供对模型对应的图形控件Figure的管理。GEF中定义了图形视图和树视图两种视图。图形视图显示各模型对应的Figure，而树视图负责显示本地树节点。

★控制器 (EditPart)：负责处理模型和视图间的交互，包含一系列编辑规则 (EditPoticy)，这些规则负责处理大部分的编辑任务。

★视图控件 (Figure/Treeitem)：用户能够看到的控件，主要用来图形化显示模型。

★编辑规则 (EditPoticy)：负责处理来自其他工具和用户的事件，创建相应的命令。

★命令 (Commands)：命令虽然不能算作模型，但与模型有着紧密的联系。命令包含模型如何被修改以及如何Undo,Redo操作信息。通常命令只对模型产生影响。

22.1.2 GEF工作原理

GEF应用的运行过程具体描述如下所示。

如图22-6所示，首先用户对编辑器中某节点的操作（如节点的选择、新增或删除等SWT事件）

被EditPartViewer捕获,并被GEF工具转换为Request,每个节点都对应一个EditPart对象。然后,用户发出的Request被转发给安装在EditPart内的一个EditPolicy组,由GEF选择适当的EditPolicy,最后由EditPolicy会返回一些Command对象来对模型做相应修改,这些执行的Command会保留在EditDomain的CommandStack里。模型发生变化时,会通知作为监听者的EditPart, EditPart再刷新相应图形以保持信息的同步。

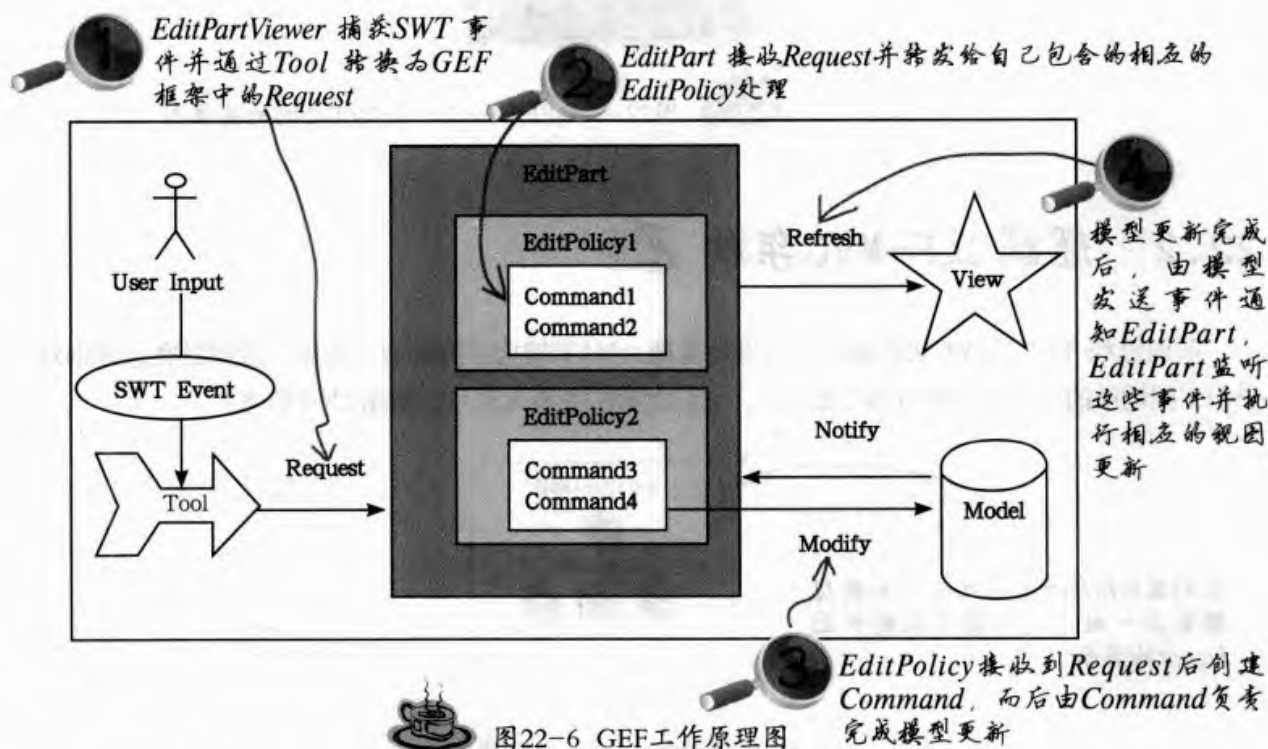


图22-6 GEF工作原理图

创建GEF应用的编辑器通常是继承Eclipse的编辑器 (IEditorPart), Eclipse的不少其他子项目也都需要这个类的支持。它允许开发人员以图形化的方式展示和编辑模型,从而提升用户体验。IEditorPart编辑器可以使开发者方便的实现drag&drop、undo&redo、move、delete、resize等图形编辑器的基本功能。GEF框架依赖于IEditorPart编辑器,因此它非常适合用于开发流程编辑器、UML图编辑器、workflow编辑器,甚至是开发像HTML这种所见即所得的文本编辑器。

由于GEF图形表现要依靠SWT和Eclipse的 Views,而Draw2d是GEF绘制图形的基本工具,因此GEF可被用于工作台中任何可以用SWT控件的地方,如一个编辑器,一个视图,一个向导页等。GEF应用一般包括一个编辑器,一个属性视图和一个大纲视图。其中编辑器是绘制和编辑图形的主要工作区,它可以带有一个放置各种工具的调色板 (Palette)。

图22-7描述了Draw2d,GEF和SWT之间的关系。GEF需要Eclipse RCP和Views插件(org.eclipse.ui.views),Views提供了属性页的支持。Draw2d主要定位于图形的绘制和布局,而GEF在Draw2d的基础上增加了编辑动作,它的主要目的如下所示。

- ★使以Draw2d图形显示模型变得容易。
- ★支持来自鼠标、键盘,或者工作台的交互动作。
- ★提供上述功能相关的通用组件。

RCP - Rich Client Platform
即富客户端平台

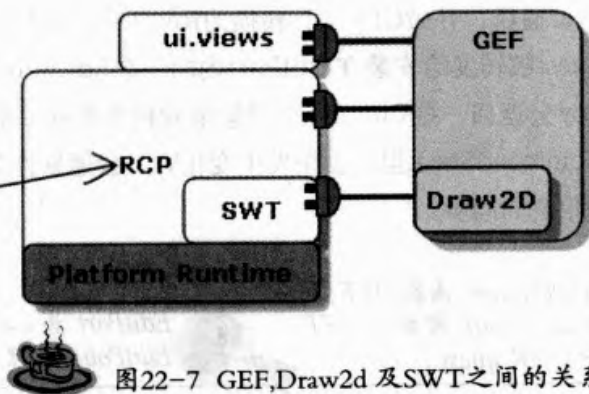


图22-7 GEF, Draw2d 及 SWT 之间的关系图

22.2 理解 GEF-MVC 架构

前面已经介绍了 MVC 架构是 GEF 的设计基础。MVC 架构中的模型、视图、控制器概念分别对应于 GEF 架构中的 Model, Figure 和 EditPart。它们之间的关系大致可以用图 22-8 描述。

控制器 EditPart 负责把视图和模型联系在一起，根据模型控制视图 Figure 的显示。

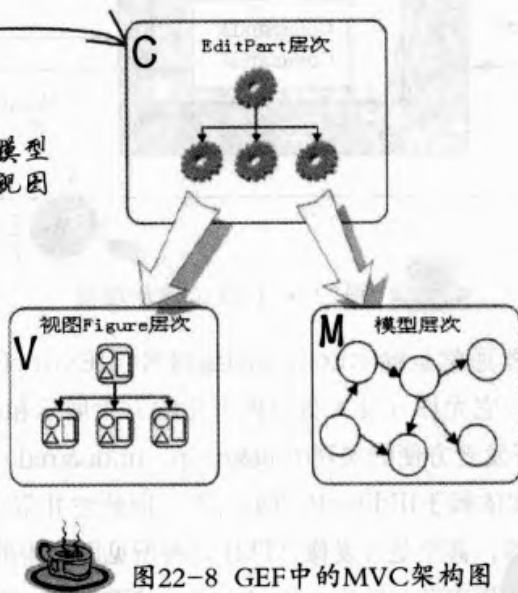


图22-8 GEF 中的 MVC 架构图

为了理解 GEF 架构，必须要搞清楚 GEF 中的模型、视图和控制器，下面就首先介绍 GEF 架构中的这三个核心组成部分。

22.2.1 GEF 中的模型

GEF 中模型通常是用户创建的需要图形编辑器编辑的对象，它要由开发人员根据需要设计出模型结构和各个模型的属性。模型的创建一般都以手工完成，但在有些情况下也常采用 EMF (Eclipse Modeling Framework) 来构造模型，EMF 是 Eclipse 平台中的建模工具，它具有直接生成模型代码的功能。由于建模不是本章的重点，在这里就不作介绍了。

在构建GEF应用时，模型可以通过实现IPropertySource接口配合属性视图中结点属性的显示和编辑。有时为了降低模型的属性维护成本，也可以通过构造一些抽象模型类（例如让所有具有连接的模型继承Node类）将一些模型通用属性集中管理和维护。图22-9描述了FNode和FConnection模型继承实现了Cloneable, Serializable, IPropertySource 接口的FElement，FElement可以包含FNode和FConnection的公共属性以及一些通用方法的实现。

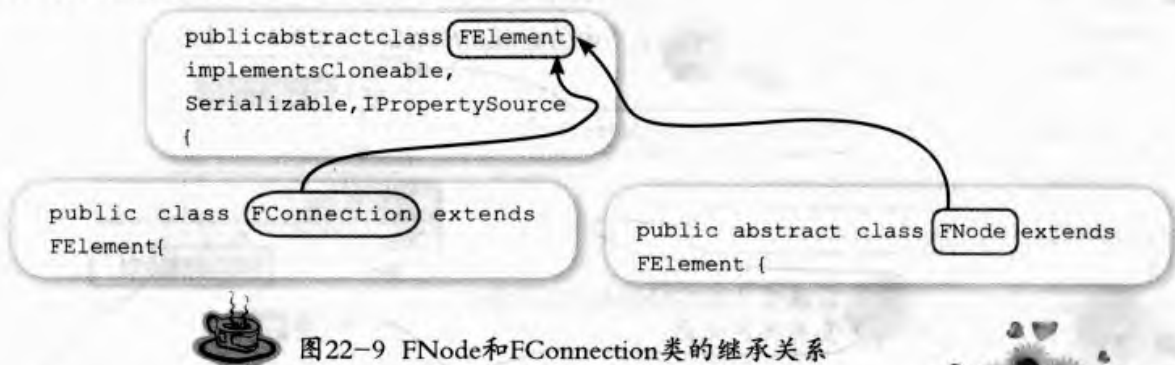


图22-9 FNode和FConnection类的继承关系

22.2.2 GEF中的视图

GEF中定义的视图为EditPartViewer，它是模型对应图形元素Figure的容器。EditPartViewer将GEF控制器无缝集成到Eclipse工作台，并负责管理GEF中各个EditPart的整个生命周期。EditPartViewer在GEF应用中所起的作用和JFace中的Viewer十分类似，EditPart就相当于JFace视图中的ContentProvider和LabelProvider。

GEF的EditPartViewer视图实现有很多种，目前提供的最常用的视图有图形（GraphicalViewer）和树状（TreeView）这两种，前者利用Draw2d图形（IFigure）作为表现方式，多用于编辑区域；后者则多用于实现大纲视图。视图虽然只负责图形元素的显示，但它的任务同样繁重。视图除了负责显示模型对应的图形元素Figure以外，还要提供编辑功能、反馈（Feedback）、工具提示（ToolTip）等。

视图EditPartViewer实现了ISelectionProvider接口，如下面代码所示，这样当用户在编辑区域做选择操作时，注册的SelectionChangeListener就可以收到选择事件。

```

public interface EditPartViewer
    extends org.eclipse.jface.viewers. ISelectionProvider
  
```

GEF提供了两种GraphicalViewer默认实现，ScrollingGraphicalViewer和GraphicalViewerImpl，它们的主要区别是前者支持滚动而后者不支持，当然也可以通过设置实现ScrollingGraphicalViewer而不显示滚动条。

在创建GEF应用时经常使用的Editor是GraphicalEditorWithPalette（是EditorPart的子类，具有图形化编辑区域和一个部件面板），这个编辑器使用GraphicalEditViewer和PaletteViewer这两个视图类，PaletteViewer也是GraphicalEditViewer的子类。开发人员要在视图类接口中的configureGraphicalViewer()和initializeGraphicalViewer()这两个方法里对EditPartViewer进行定制，包括指定它的contents和EditPartFactory等。

在GraphicalViewer中GEF依靠Draw2d中的轻量级图形元素Figure完成界面展现。GraphicalEditPart是GraphicalViewer中维护的负责图形展现的EditPart。GraphicalEditPart中包含有一个Figure，它负责创建Figure，在模型发生改变后更新Figure和在被设为不可用后从界面中删除Figure图形。图22-10描述了模型与Figure之间的交互过程。

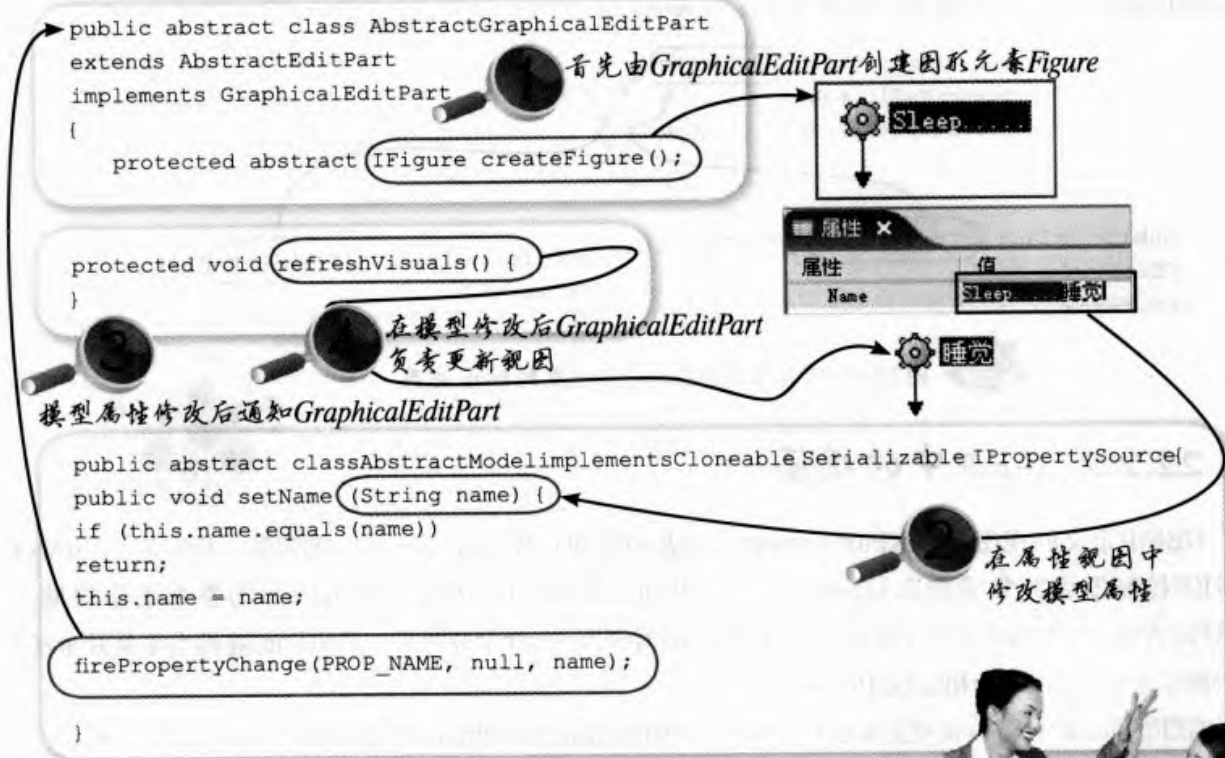


图22-10 模型与Figure之间的交互



AbstractGraphicalEditPart是GraphicalEditPart接口的默认实现，其中定义的createFigure()方法负责创建Figure。通过重写该方法，读者可以添加创建自己的Figure图形元素的代码。第21章中已经对Draw2d中的Figure作了详细介绍，读者应该已经对如何创建Figure有所了解。在调用EditPart中的createFigure()之后，可以用getFigure()方法获取已经创建的Figure。

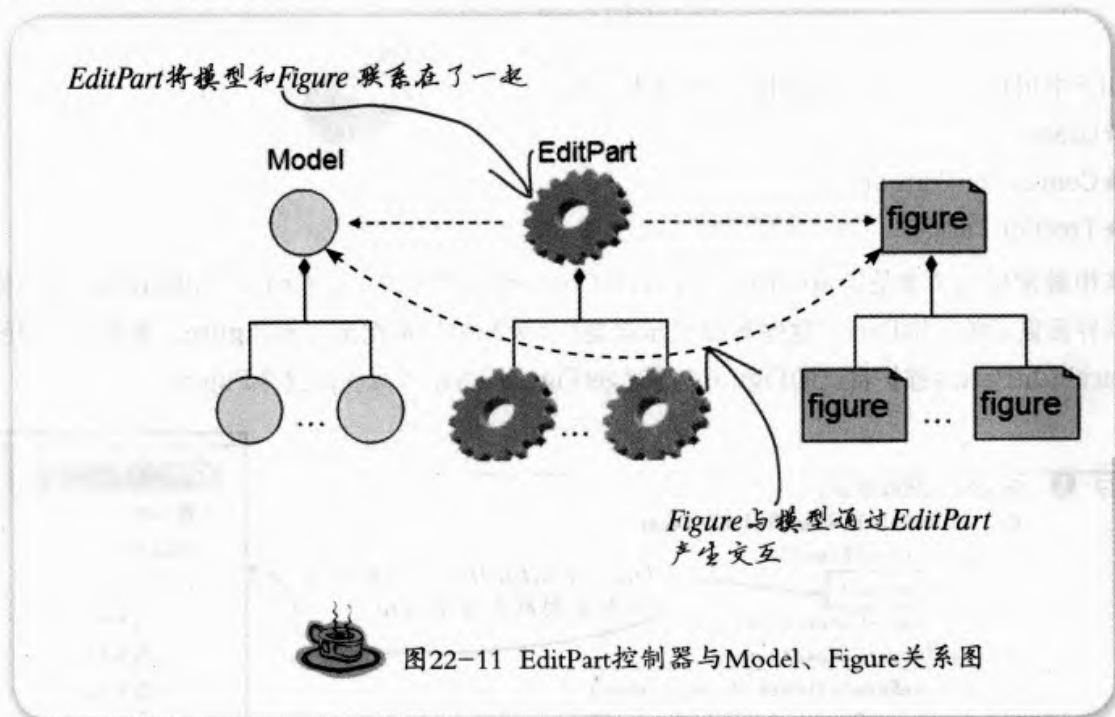
※ 注意: ※

用户自定义对应于模型的Figure图形的更新在AbstractEditPart类定义的refreshVisuals()方法中完成。如果某些模型属性修改对应图形的修改，读者还需要重写这个方法并在其中定义对应模型属性的Figure图形样式变化。



22.2.3 GEF中的控制器——EditPart

在MVC架构里控制器是模型与视图之间的媒介，它是整个GEF中的核心，同时也是最复杂的部分。EditPart不仅要监听模型的变化，而且当用户编辑视图时，还要把编辑结果反映到视图上。GEF中的控制器是被称为EditPart的对象，更确切地说应该是一组EditPart对象共同组成了GEF的控制器部分。一般情况下每一个模型对象都对应一个EditPart对象，模型与Figure之间的映射和交互由EditPart负责。如图22-11所示。



EditPart是GEF架构的控制中心，它掌管着模型元素如何与图形Figure映射和Figure在不同情况下的行为、样式定义。通常情况下，模型元素与EditPart是一一对应的，这也意味着EditPart类与模型元素类有着相同的结构。

※ 注意: ※



在GEF中org.eclipse.gef.EditPart接口定义了EditPart的基本功能和结构。org.eclipse.gef.AbstractEditPart抽象类包含了EditPart的基本实现。在创建自己的EditPart时，GEF建议直接扩展AbstractEditPart抽象类，并不需要自己实现EditPart接口。这样做的好处是避免在未来接口API发生变化时，GEF应用程序也需要修改。直接扩展抽象类可以提高程序的兼容性。

当添加EditPart到GEF应用中时，需要创建一个EditPartFactory对象负责根据给定模型对象创建对应的EditPart对象，这个工厂类将被GraphicalViewer用来创建模型对应的EditPart。如下面代码所示。


```
public interface EditPartFactory {
```

```
    EditPart createEditPart(EditPart context, Object model);
```

```
}
```

此方法根据传入的模型参数创建对应的EditPart并返回

GEF中用到的EditPart大致可以分为以下三类。

- ★GraphicalEditPart
- ★ConnectionEditPart
- ★TreeEditPart

其中最常用的两类是GraphicalEditPart和ConnectionEditPart。GraphicalEditPart是为模型元素提供界面显示的EditPart，这些界面显示就是Draw2d的轻量图形元素Figure。如图22-12所示，GraphicalEditPart内维护自己的Figure，通过getFigure()方法可以获取这个Figure。

```
GraphicalEditPart
+ addNodeListener(NodeListener)
+ getContentPane()
+ getFigure()
+ getSourceConnections()
+ getTargetConnections()
+ removeNodeListener(NodeListener)
+ setLayoutConstraint(EditPart, IFigure, Object)
```

GraphicalEditPart 与其对应的
界面图形元素 Figure

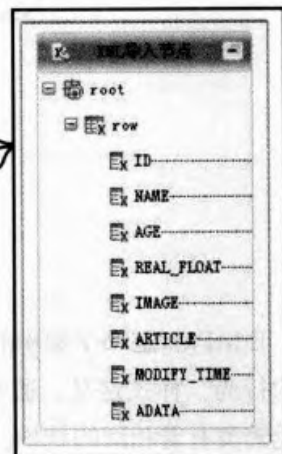


图22-12 GraphicalEditPart和Figure

ConnectionEditPart是负责控制连接GraphicalEditPart的连线EditPart。如图22-13所示，createFigure()方法负责创建出连线Figure，接着ConnectionEditPart通过getSourceConnectionAnchor()和getTargetConnectionAnchor()方法得到连线所连接的起始锚点和终止锚点，并在图形界面上绘出连线图形。

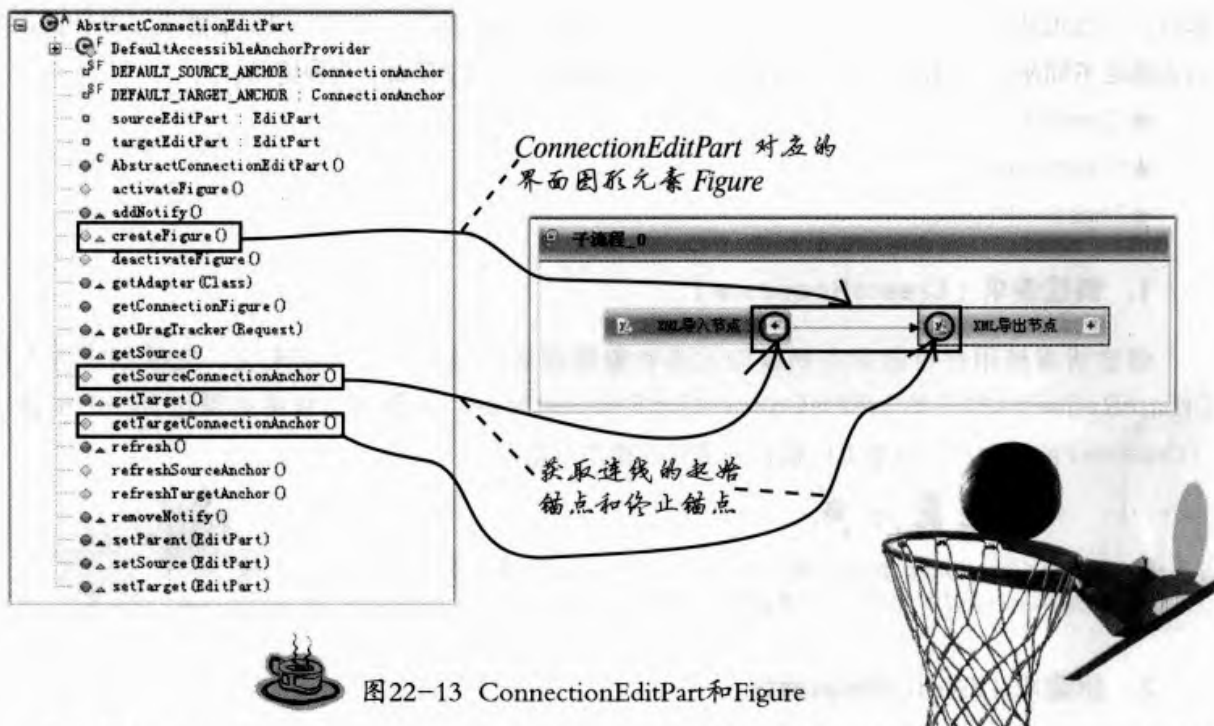


图22-13 ConnectionEditPart和Figure

第三种EditPart是TreeEditPart，它只在用SWT tree控件创建树形结构的模型界面显示时用的到。

EditPart接口中提供了大量的方法，在一般情况下，需要调用的只有getModel()和setModel()方法，其他的方法大多是GEF框架为了维护EditPart自己调用的。这些方法已经提供了默认实现，也可以根据需要重写这些方法或添加自己的方法。如可以重写createFigure()方法创建模型对应的Figure或在EditPart添加维护模型元素和模型属性的方法。

EditPart是有生命周期的，在被创建以后并不是立即可用（active）的，只有在GEF得到通知以后，该EditPart才被设为可用状态。在某些情况下，如编辑器（editor）被关闭或EditPart对应的模型元素被删除的时候EditPart就会过期，这时候GEF已经不再需要这些EditPart，它们就会被设为不可用（deactivated）。在EditPart状态发生改变时，GEF框架就会调用EditPart中的activate()和deactivate()方法。通过调用isActive()方法可以获取EditPart的当前状态。当EditPart已经过期后，也可以被重新设为可用状态。

22.3 GEF中的基本概念

在利用GEF创建GEF应用的过程中，还需要了解一些其他的基本概念，下面就对这些概念做一下详细介绍。

22.3.1 Request和Command

请求（Request）和命令（Command）是GEF中界面操作与GEF交互操作中通信与执行动作的基

本概念。请求是GEF中用来通信的对象，它包含有执行请求所需的信息。GEF中提供了多种请求类型用来满足不同使用场合的需要。在创建GEF应用时最常用到的有如下几种请求。

- ★CreateRequests
- ★GroupRequests
- ★LocationRequests

1. 创建请求 (CreateRequests)

创建请求被用在可能会有新模型元素对象被创建的地方。对连线来说，它的创建请求就是CreateRequests的子类CreateConnectionRequest对象。在创建请求中必须包含一个创建工厂(CreationFactory)，创建工厂负责创建新的模型对象。



注意：

在实现CreationFactory时候，并不一定要包含创建模型对象代码，在请求对应的Command中也可以创建模型对象。

2. 组请求 (GroupRequests)

组请求是可以传递给多个EditPart的请求类型。典型的组请求是ChangeBoundsRequests，它在EditPart被移动或被改变大小时被创建。

3. 位置请求 (LocationRequests)

位置请求负责跟踪一个位置的变化，如SelectionRequest，它在EditPart被选中时创建。读者可以自己定义EditPart的选中方式，这样在单击EditPart的不同位置的时候就能够完成不同的功能。

图22-14中演示了请求从创建到处理流向全过程。首先请求创建者（如一些工具(tool)、动作(action)或拖放管理器）创建请求，然后请求被传递到EditPart。EditPart其实并不负责处理请求，它只负责把请求发送给对应的EditPolicy，EditPolicy负责“翻译”请求信息，并为请求创建命令，由命令来完成请求任务以及Undo/Redo请求操作。

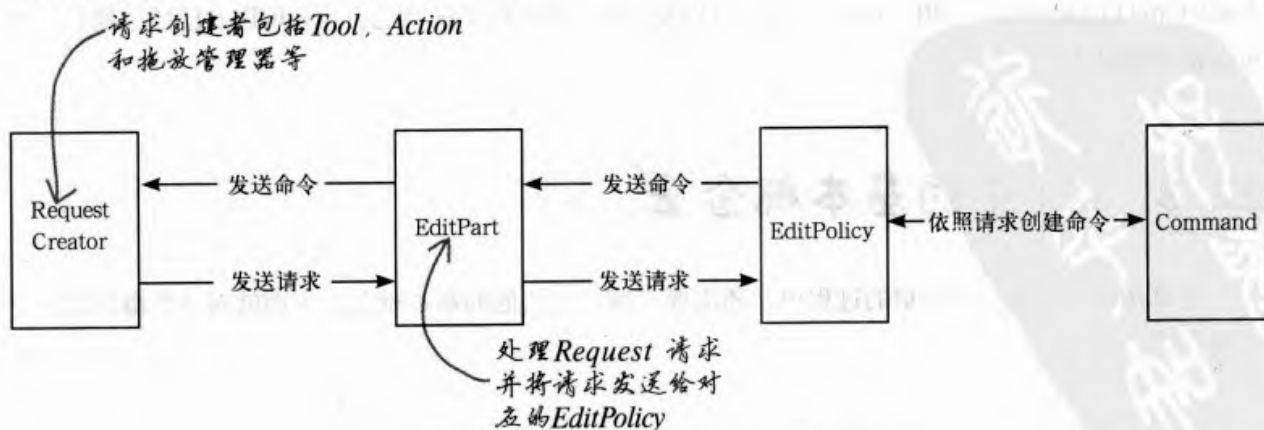


图22-14 请求处理过程示意图

GEF中请求的类型是由类型常量区分的，类型常量定义在RequestConstants接口中，当EditPart接收到请求后，会根据请求不同的类型使用不同的策略处理。Request类中定义了getType()方法，通过它能够获取请求的类型。图22-15描述了请求类型在请求处理过程中的作用。



图22-15 请求中的类型

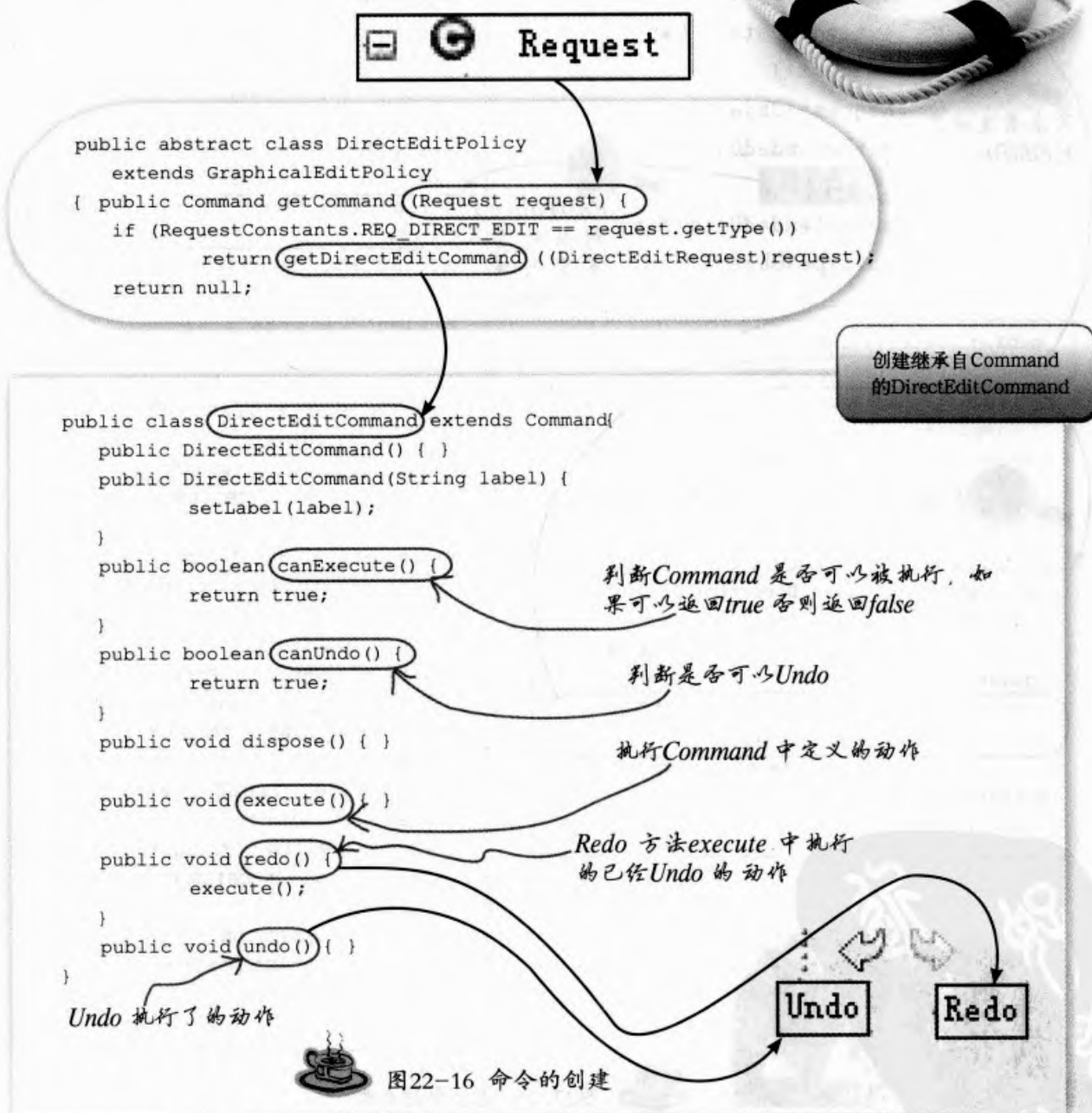
RequestConstants接口中按请求产生的方式为不同的请求定义了不同的类型常量值，EditPart按照这个值区分请求。

在EditPolicy接收到指定类型的请求时，它需要根据请求中附带的信息创建命令，由命令完成请

求动作。命令在GEF中是可以直接修改模型的部分。命令的主要职责包括以下几点。

- ★执行请求合法性验证;
- ★执行Undo/Redo操作;
- ★执行指定动作。

图22-16描述了命令的创建过程和命令中常用方法的作用。



※ 注意: ※

每当需要在应用中加入对某种请求的处理时, 都需要定义命令。GEF中提供的包含命令基本实现的抽象类为 `org.eclipse.gef.commands.Command`, 在实现自己的命令时需要直接扩展此抽象类。



22.3.2 RootEditPart

GEF自带的EditPart中，RootEditPart算是最特殊的一种。它不同于一般的EditPart，不对应任何模型对象，也不起一般控制器的作用。RootEditPart的作用是把EditPartViewer和Contents（GEF应用的最下层EditPart，一般代表绘制图形的画板或背景）联系起来，它们为EditPart提供运行环境，可以把它理解为处于GraphicalViewer和EditPart之间的交互接口。EditPartViewer包含方法setRootEditPart()专门用来指定视图对应的RootEditPart。

常用的RootEditPart有如下几种。

- ★ScalableRootEditPart
- ★FreeformGraphicalRootEditPart
- ★ScalableFreeformRootEditPart

这三种RootEditPart都必须用在ScrollingGraphicalViewer视图中，因此，它们也都支持滚动条功能。在这里Freeform图形编辑的意思是在图形编辑区域拖动Figure到图形边界的时候，编辑区域会向该边界方向自动扩展图形。这种功能对图形元素比较多而且需要自由摆放图形中图形元素的使用场合是很有用的，但是如果图形元素需要放置在固定的位置，如放在表格中，那么Freeform图形编辑就不适用了。表22-1中分别对这三种RootEditPart的特性进行了对比。

表22-1 三种RootEditPart特性对比表

RootEditPart	主Figure	是否Freeform	是否Scalable
ScalableRootEditPart	Viewport	否	是
FreeformGraphicalRootEditPart	FreeformViewport	是	否
ScalableFreeformRootEditPart	FreeformViewport	是	是

RootEditPart是编辑器中所有EditPart的根，它把EditPartViewer和EditPart连接在了一起。GEF中提供了一些RootEditPart的默认实现。图22-17描述了创建RootEditPart和GraphicalViewer并将RootEditPart设置到GraphicalViewer之中的过程。

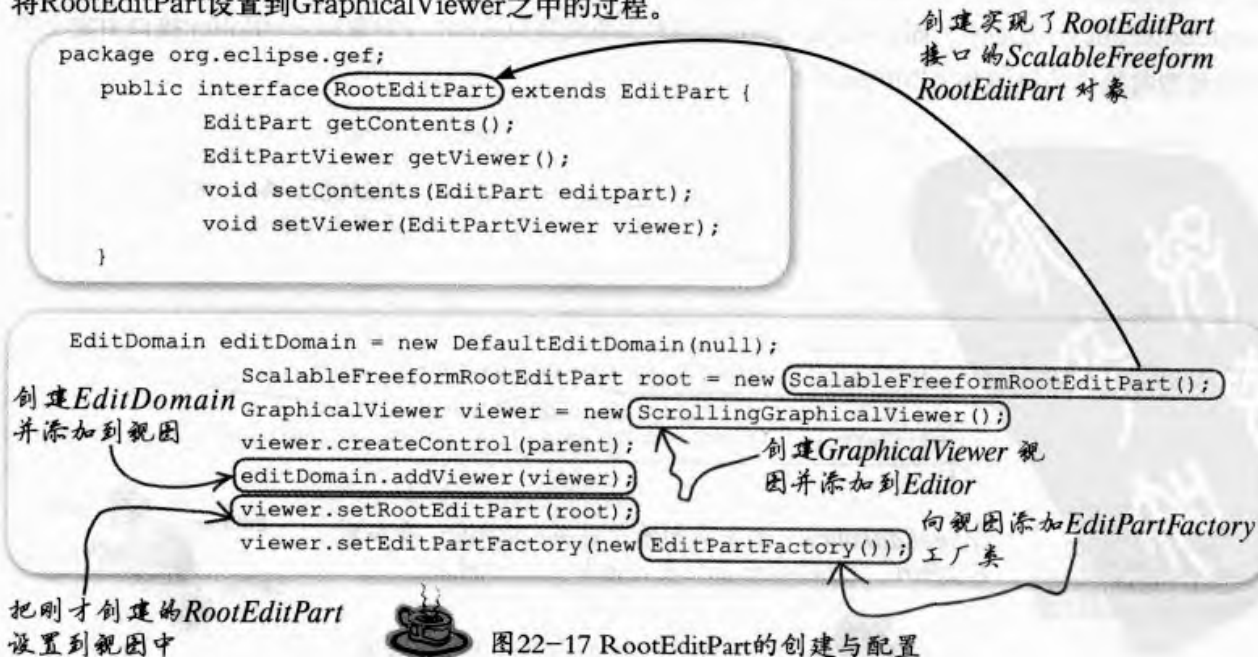


图22-17 RootEditPart的创建与配置

22.3.3 EditPolicy 和 Role

通过对请求Request和命令Command的介绍,读者应该已经对GEF中的请求处理机制有了一个大体的了解,并且知道EditPolicy是请求的最终执行者和Command的创建者。下面来探寻请求的执行过程和EditPolicy的实现原理。

策略 (EditPolicy)

图22-18列出了EditPolicy接口的定义,所有自定义的EditPolicy都要实现这个接口。在EditPolicy接口中定义了Role的类型常量

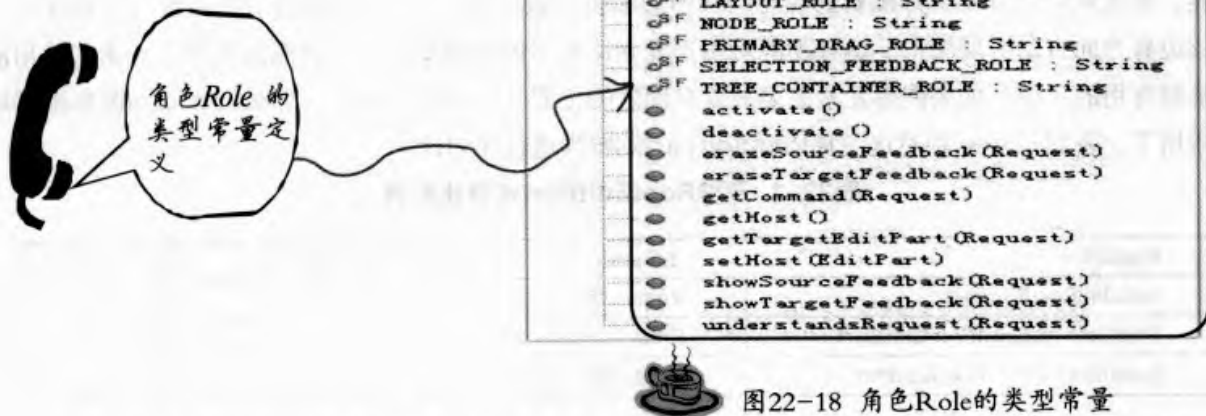


图22-18 角色Role的类型常量

图22-19描述了StepModel模型对应的EditPart - StepPart类中EditPolicy的定义。当为自己定义的模型创建对应的EditPart时,EditPart中所要处理的请求对应的EditPolicy需要添加在createEditPolicy()方法中,而installEditPolicy()方法负责将EditPolicy对象以EditPolicy接口中定义的Role类型常量作为键值加入EditPart中。

```

public class NodePart extends AbstractGraphicalEditPart implements
PropertyChangeListener, NodeEditPart {

    protected void createEditPolicy() {

        installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new NodeDirectEditPolicy());

        installEditPolicy(EditPolicy.COMPONENT_ROLE, new NodeEditPolicy());

        installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new NodeGraphicalNodeEditPolicy());

    }
}

```

为EditPart添加EditPolicy

createEditPolicy() 方法在初始化时将EditPolicy 添加到EditPart, EditPolicy 与角色 (Role) 一一对应



图22-19 StepPart类中EditPolicy的定义

EditPart是请求的接收者而EditPolicy则是请求的具体执行者，当EditPolicy接收到请求后它会判断请求类型并分配给对应的EditPolicy执行。

下面详细介绍几种常用的EditPolicy策略。

1. 图形节点编辑策略 (GraphicalNodeEditPolicy)

图形节点编辑策略提供了在创建连线或改变连线连接位置时提供图形显示反馈支持。当一个连线连接目标节点改变的时候，它会与继承自NodeEditPart的EditPart一起提供一个虚拟连接。NodeEditPart会根据当前鼠标位置返回最佳锚点位置，而后画在feedback层上的虚拟连接就会连接到该锚点。一般情况下该锚点是离当前鼠标位置最近的连接源端锚点。策略中允许自定义的功能点是重写createDummyConnection()方法，以返回自定义的表示新建连线的连线Figure。一般是不同于已有连线的颜色、粗细等图形表现样式的连线。

2. 布局编辑策略 (LayoutEditPolicy)

当EditPart需要使用布局管理器管理子EditPart的时候，布局管理器就要根据layout constraint计算各个子EditPart的位置，并通过LayoutEditPolicy的子类编辑策略更新图形显示。LayoutEditPolicy中提供的主要方法包括以下两种。

✎ showLayoutTargetFeedback：这个方法会对当前操作的目的Figure产生界面图形上的反馈。子类可以重写这个方法控制或调整Figure所在的位置。这个方法返回的光标Figure会指示操作位置信息。

✎ getSizeOnDropFeedback：当拖动操作完成后，该方法将显示调整大小后的Figure。
允许自定义的功能有以下两种。

✎ 默认情况下showLayoutTargetFeedback方法不做任何事情，所以需要在子类中加入显示插入新对象位置的代码。

✎ getSizeOnDropFeedback可以被改写显示各种形状、颜色等。当需要限制Figure的大小时，也可以重写这个方法。

3. FlowLayout布局编辑策略 (FlowLayoutEditPolicy)

这个策略主要被用在EditPart对应的Figure采用FlowLayout布局管理器的情况下。这个类中提供了两个像素宽的接入标识线条。通过类中的方法getLineFeedback()可以获取默认的连线Figure，这样就可以修改连线的某些属性，如线条的粗细、样式等。如果需要使用其他Figure，那么就可以考虑重写showLayoutTargetFeedback()方法来计算新Figure的大小和位置。

4. 选择编辑策略 (SelectionEditPolicy)

如果某个EditPart被选中，这个策略就负责提供选中动作的图形反馈。需要注意的是反馈中的Figure会被画在feedback layer (LayerConstants.FEEDBACK_LAYER)，SelectionEditPolicy是所有选中策略的抽象基类，类中的主要方法包括以下几个。

★protected void showFocus()

★protected abstract void showSelection()

★showPrimarySelection()

★hideFocus()

★hideSelection()

从方法的名称就能了解这些方法各自的用途。继承自该策略的子类可以自定义的功能有以下几种。

★为非矩形图形元素提供自定义的焦点获取或选中状态标识。

★实现只能选中而不能被移动的图形元素。

★定义特殊的选中状态标识。

5. 选择控制点编辑策略 (SelectionHandlesEditPolicy)

此策略是支持带控制点的选中策略，它的子类会提供选中EditPart周围的所有控制点列表。GEF中提供的SelectionHandlesEditPolicy子类有以下几个。

★BendpointEditPolicy：在选中连线时，能显示转折点控制点的策略。

★ConnectionEndpointEditPolicy：如果需要改变连线连接位置，这个策略能够在选中连线时显示连线端点控制点。

★NonResizableEditPolicy：不支持改变选中EditPart的策略，它会在选中EditPart周围显示一个边框，并在每一个角上添加一个控制点用来拖动EditPart。

★ResizableEditPolicy：支持改变选中EditPart大小的策略，这个类扩展自NonResizableEditPolicy，并在选中矩形边框的各边线的中点加上了调整选中EditPart大小的控制点。

角色 (Role)

用户在对图形编辑过程中产生的SWT事件会被GEF转换为一系列请求，GEF定义了很多种类的请求，如REQ_RESIZE表示用户改变对象的大小，REQ_DIRECT_EDIT表示直接编辑某对象。接口RequestConstants定义各种请求的含义，这些请求在EditPart中与负责处理他们的EditPolicy的对应关系被称为角色 (Role)。角色又被划分为图形化(Graphical)角色和非图形化(NonGraphical)角色两大类，前者如SELECTION_FEEDBACK_ROLE对应图形节点选择后的反馈操作，而后者包括CONNECTION_ROLE对应和连接有关的操作等。其他的角色请参看接口EditPolicy中的定义。角色这个概念是通过编辑策略来实现的，EditPolicy为EditPart增加了编辑功能，其主要功能是根据请求创建相应的命令，命令会直接操作模型对象。每一个EditPart可以对应不同角色安装 (install) 多个EditPolicy，用户对这个EditPart的操作会被GEF交给已安装的对应该EditPolicy处理。这样可以在不同EditPart之间通过EditPolicy共享一些交互处理。

EditPolicy在GEF中把特定的请求处理封装在了一起，并可以按角色加入EditPart中。这是一种基于良好面向对象设计方案。EditPart中如果没有EditPolicy将不能处理任何编辑任务，包括选中功能都无法实现。EditPolicy也是按角色来分类的。接口org.eclipse.gef.EditPolicy中预定义的角色常量如下所示。

★COMPONENT_ROLE：组件角色

★CONNECTION_B.ENDPOINTS_ROLE：连接折点角色

- ★CONNECTION_ENDPOINTSROLE: 连接终点角色
- ★CONTAINER_ROLE: 容器角色
- ★DIRECT_EDIT_ROLE: 直接编辑角色
- ★GRAPHICAL_NODE_ROLE: 图形编辑角色
- ★LAYOUT_ROLE: 布局角色
- ★NODE_ROLE: 图形节点角色
- ★PRIMARY_DRAG_ROLE: 主拖动角色
- ★SELECTION_FEEDBACK_ROLE: 选择反馈角色
- ★TREE_CONTAINER_ROLE: 树容器角色

注意: 

*EditPart*中一种角色只能够包含一个*EditPolicy*。

下面介绍几种最常用的角色类型的功能和使用范围。

1. Component_role

组件策略角色对应的EditPolicy的基类为ComponentEditPolicy。它是负责包含模型元素的EditPart的基本操作（如删除模型元素等）的主要策略角色。当需要直接修改EditPart的模型的某些与图形表现无关的属性请求时，可以考虑采用ComponentEditPolicy来处理。

2. Connection_role

连接策略角色的基类为ConnectionEditPolicy，它其实是对应于ConnectionEditPart的专用的组件策略角色。

3. Container_role

容器策略角色的基类为ContainerEditPolicy，它负责包含子EditPart的容器EditPart的操作处理，每一个支持编辑的容器EditPart都需要一个ContainerEditPolicy。

4. Layout_role

布局策略角色的基类为LoayoutEditPolicy，它负责包含布局定义的容器EditPart的布局管理任务（其中包括处理需要计算各个子EditPart位置坐标的请求）。这里需要注意的是在处理位置改变请求时，ContainerEditPolicy和LoayoutEditPolicy在职责上有些重叠，ContainerEditPolicy只适用于不需要关心其所包含的子EditPart位置的场合下，在ContainerEditPolicy中不包含任何可用的坐标信息。

5. Tree_container_role

树容器策略角色的基类为TreeContainerEditPolicy，它是对应于TreeEditPart的专用容器策略角色。

6. Graphical_node_role

图形节点策略角色的基类为GraphicalNodeEditPolicy，它负责需要在节点之间创建连接的

EditPart中连接的创建和管理请求处理。当需要在EditPart之间创建连接时，就需要在其中加入GraphicalNodeEditPolicy策略。

7. Direct_edit_role

直接编辑策略角色的基类为DirectEditPolicy，它可以使EditPart具有直接编辑的功能。当双击具有此策略角色的EditPart时，可以直接在图形上键入修改指定的属性。

22.3.4 图形视图——GraphicalViewer

GEF的图形显示依赖于轻量级图形系统Draw2d。在第21章中，已经介绍了图形元素Figure是由轻量系统类LightweightSystem绘制的，但是，LightweightSystem并不能完成所有的Figure维护工作。当创建自己的基于GEF的Eclipse编辑器的时候，有一些组件并不需要自建应用来维护，这些维护工作就会由GraphicalViewers完成。

视图是Eclipse平台编辑器的主要部分，它的默认实现是GraphicalViewer类。添加视图最好是在createPartControl()方法中完成。添加过程一般是首先创建一个GraphicalViewer实例，而后配置这个视图实例，并加入EditDomain中。

GEF中预定义了两种GraphicalViewer，一种是ScrollingGraphicalViewer，它支持滚动条操作，另一种是GraphicalViewerImpl，它不支持滚动条操作。在实现应用时，通常会用ScrollingGraphicalViewer，这种视图也可以隐藏视图中的滚动条。值得一提的是GraphicalViewer本身相对独立，它并不需要依赖Eclipse Editor。GraphicalViewer中包含有无参数的构造函数和为视图添加SWT控件的createControl()方法，因此，它可以被用在任何允许创建SWT控件的地方。在视图和视图中包含的控件创建完成后，读者还需要调用EditPartViewer中的setRootEditPart()，setEditPartFactory()和setContent()方法，分别向视图添加RootEditPart，EditPart工厂类EditPartFactory和Content。Content是所有模型元素的根模型。

GraphicalViewers之中包含有注册EditPart的注册器（Registry），读者可以通过调用方法getEditPartRegistry()获取。在注册过程中，EditPart的注册键（key）需要读者设置。GEF中也提供了一个默认的实现，它可以自动完成注册或注销所有视图中包含的EditPart，并把EditPart对应的模型作为注册键。

在创建JFace视图的时候，一般情况下只需要Content，一个工厂类和一些配置就完成了。视图的一些复杂功能如拖动操作、事件和更新处理并不需要添加就已经有了。GraphicalViewer在GEF中完成的也是类似的功能。图22-20描述了GraphicalViewer在GEF中的运行原理。

```
public class PartFactory implements EditPartFactory {
    public EditPart createEditPart(EditPart context, Object model) {
        EditPart part = null;
        if (model instanceof Diagram)
            part = new DiagramPart();
        else if (model instanceof ConnectionModel)
            part = new ConnectionPart();
    }
}
```

建立EditPart
工厂方法

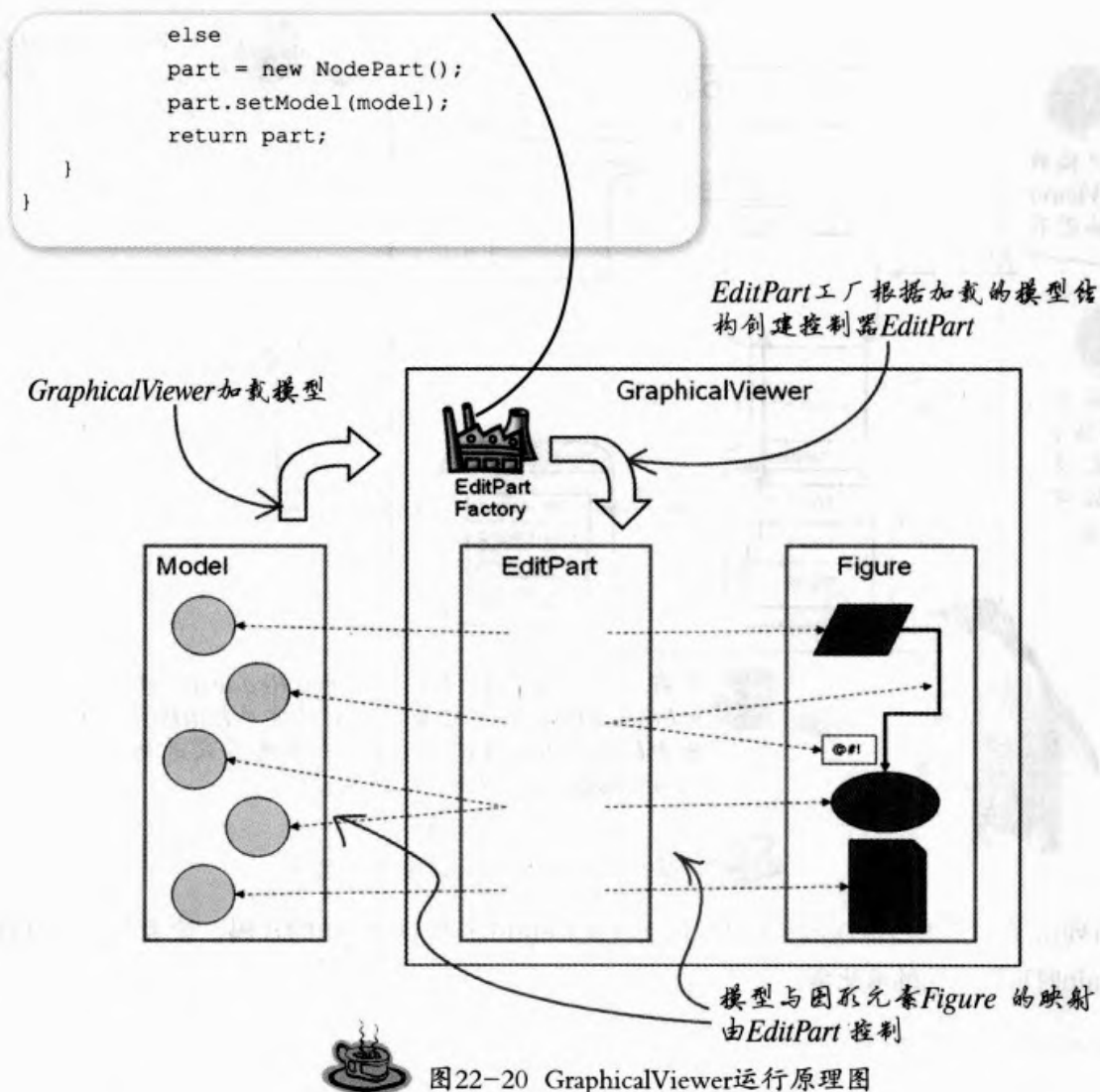


图22-20 GraphicalViewer运行原理图

※ 注意 : ※

开发人员要在`configureGraphicalViewer()`和`initializeGraphicalViewer()`这两个方法里对`EditPartViewer`进行定制，包括指定它的`contents`和`EditPartFactory`等。



22.3.5 EditDomain和CommandStack

GEF中还有两个不得不提的概念，`EditDomain`和`CommandStack`。`EditDomain`是将编辑器、视图和工具逻辑上捆绑在一起的接口，其中定义了真正的编辑器应用。`EditDomain`中提供了一个`CommandStack`。`CommandStack`负责记录`Editor`中所有执行过的命令，这使实现编辑动作的redo,undo功能成为可能，而且可被用来判断模型是否被修改。通常一个编辑器包含一个`EditDomain`，但一个多页编辑器中的几个编辑器也可以共享一个`EditDomain`。图22-21描述了`EditDomain`在GEF应用中所起的作用。

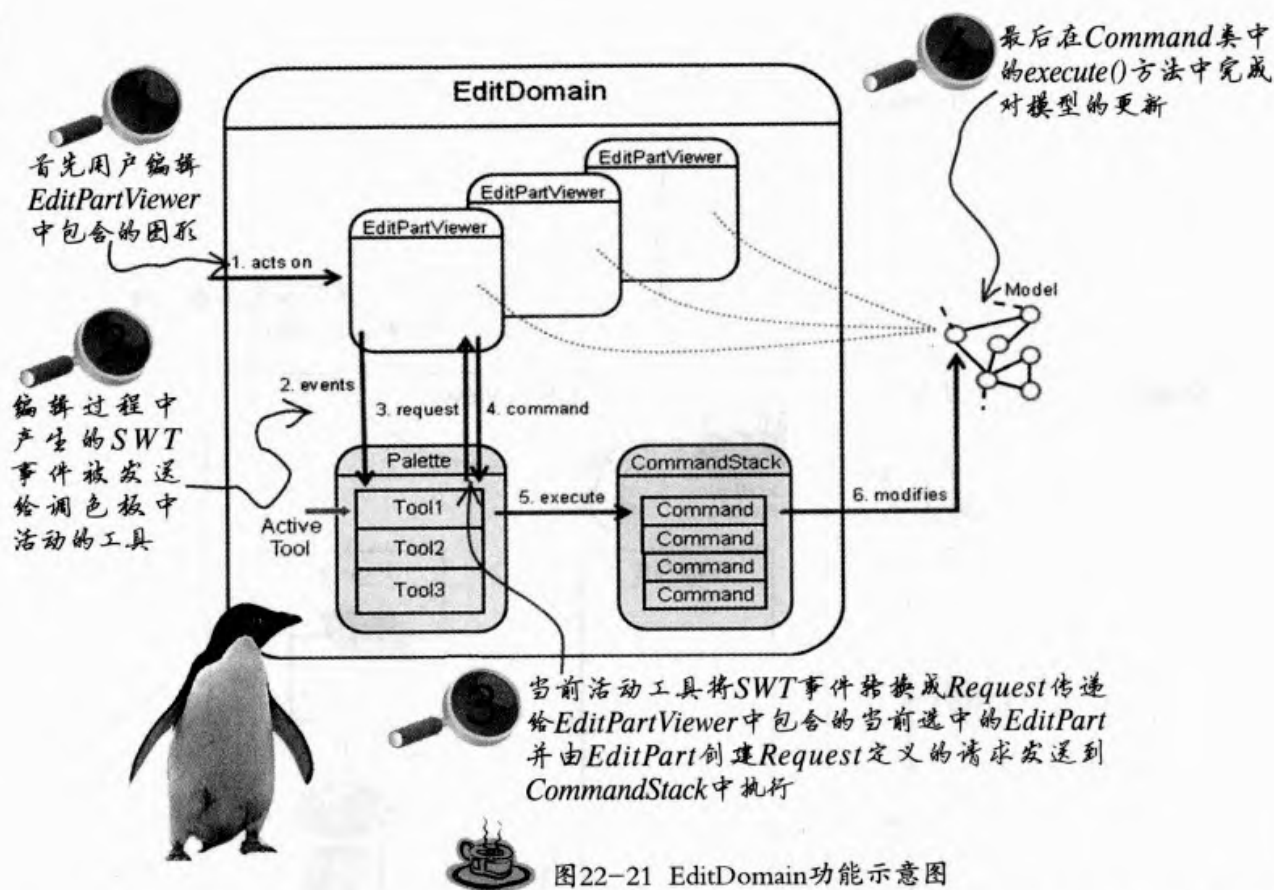


图22-21 EditDomain功能示意图

下面列出了GEF中EditDomain的默认实现类DefaultEditDomain的代码。读者在创建自己的EditDomain时可以直接继承此类。

```
package org.eclipse.gef;

import org.eclipse.ui.IEditorPart;
public class DefaultEditDomain
    extends EditDomain
{
    private IEditorPart editorPart;
    public DefaultEditDomain(IEditorPart editorPart) {
        setEditorPart(editorPart);
    }
    public IEditorPart getEditorPart() {
        return editorPart;
    }
    protected void setEditorPart(IEditorPart editorPart) {
        this.editorPart = editorPart;
    }
}
```

在初始化EditDomain 时需要为它设置EditorPart

命令堆栈 (CommandStack) 是EditDomain中提供的维护并执行各种请求对应的Command和Command执行后处理Undo/Redo已执行动作请求的对象堆栈。GEF中提供了CommandStack类的默认实现。

在创建GEF应用时，应用中的所有Command都需要在CommandStack维护执行，这样可以确保GEF中对模型的所有修改都能够被编辑器监控的到。CommandStack中提供了isDirty()方法，调用它可以知道编辑器在上次保存后，CommandStack是否执行过Command。图22-22显示了CommandStack的结构和功能。

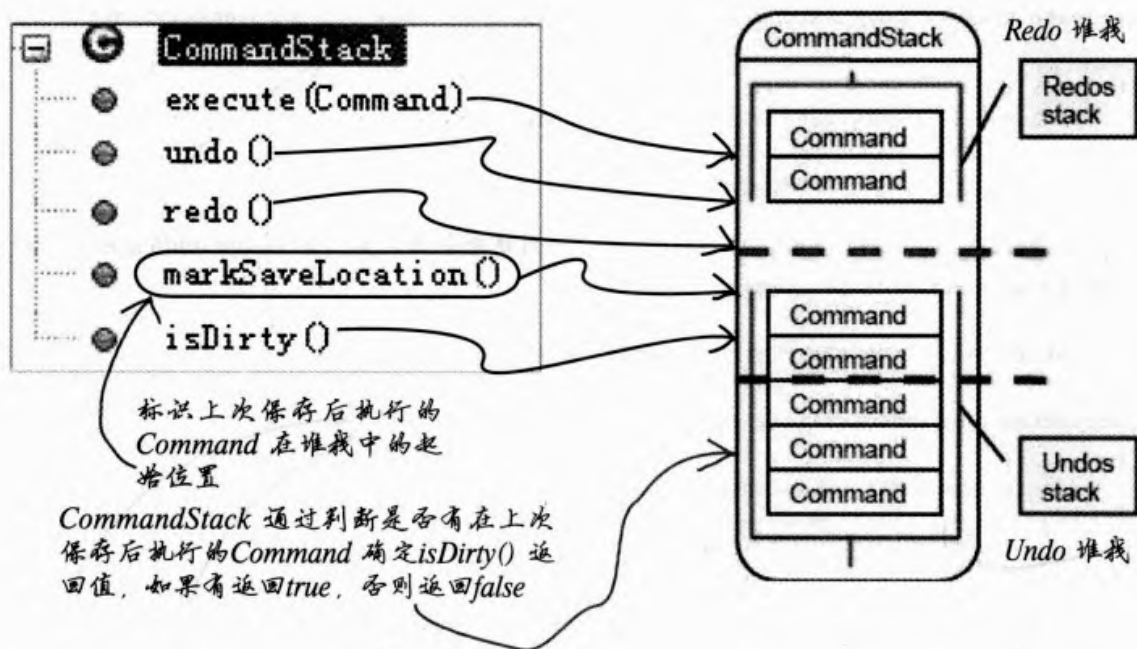


图22-22 CommandStack功能示意图

图22-23是isDirty()和setDirty()方法的默认实现。isDirty()返回布尔值，如果值为true则表示当前CommandStack中包含在上次保存后执行的Command，需要再次保存。

```
public class CommandStack{
    private boolean isDirty;
    public boolean isDirty()
    {
        return isDirty;
    }
    protected void setDirty(boolean dirty)
    {
        if(isDirty != dirty)
        {
            isDirty = dirty;
            firePropertyChange(IEditorPart.PROP_DIRTY);
        }
    }
}
```

变量isDirty 标识Editor 是否需要保存



当isDirty 被设置为true，那么CommandStack 就会通知Eclipse workbench在标题前加上星号表示需要保存修改



图22-23 CommandStack中的isDirty属性

如果要让CommandStack能够监控Editor是否保存还需要把Editor中isDirty()方法委托给CommandStack。这里还可以用监听器监听CommandStack的更新，而后修改Editor的是否需要保存的状态。在Editor状态改变后，EditorPart的父类就会通知Eclipse Workbench修改界面上的保存提示标识。代码如下所示。

```
private CommandStackListener commandStackListener = new CommandStackListener()
```

```
{  
    public void commandStackChanged(EventObject event)  
    {  
        setDirty(getCommandStack().isDirty());  
    }  
};
```

创建命令堆栈监听器 CommandStackListener

```
public CommandStack getCommandStack()  
{  
    return getEditDomain().getCommandStack();  
}
```

```
protected CommandStackListener getCommandStackListener()  
{  
    return commandStackListener;  
}
```

```
public void init(IEditorSite site, IEditorInput input)  
throws PartInitException  
{  
    setSite(site);  
    setInput(input);  
    getCommandStack().addCommandStackListener(getCommandStackListener());  
}
```

```
public void dispose()  
{  
    getCommandStack().removeCommandStackListener(getCommandStackListener());  
    super.dispose();  
}
```

向CommandStack中添加CommandStackListener

从CommandStack中移除CommandStackListener

代码描述了将CommandStackListener设置到CommandStack和将CommandStackListener移除。在编辑器Editor得到输入的时候需要向CommandStack中添加CommandStackListener，而在Editor被dispose掉时需要把CommandStackListener从CommandStack中移除。

22.3.6 调色板 (Palette) 和工具 (Tool)

在构建GEF应用中，通常情况下调色板用来显示绘图工具，当然也可以根据需要用于其他用途。在讲解如何创建工具之前，先介绍一下如何在编辑器中加入调色板。GEF中提供了调色板视图 (PaletteViewer)，在视图可以显示用来标识绘图元素的图标和说明，这些元素往往是与模型元素对应的。如图22-24所示是GEF样例编辑器中的可浮动调色板样式，这种调色板位置不固定，可以单

击左上角箭头隐藏或显示，或者拖动标题栏来改变位置。



图22-24 调色板样式



调色板模型 (Palette Model) 由PaletteRoot维护，每一个PaletteViewer都需要一个PaletteRoot。PaletteRoot是一个调色板容器，它被用来维护调色板入口 (PaletteEntry)。图22-25以树形结构描述了调色板相关类之间的关系。

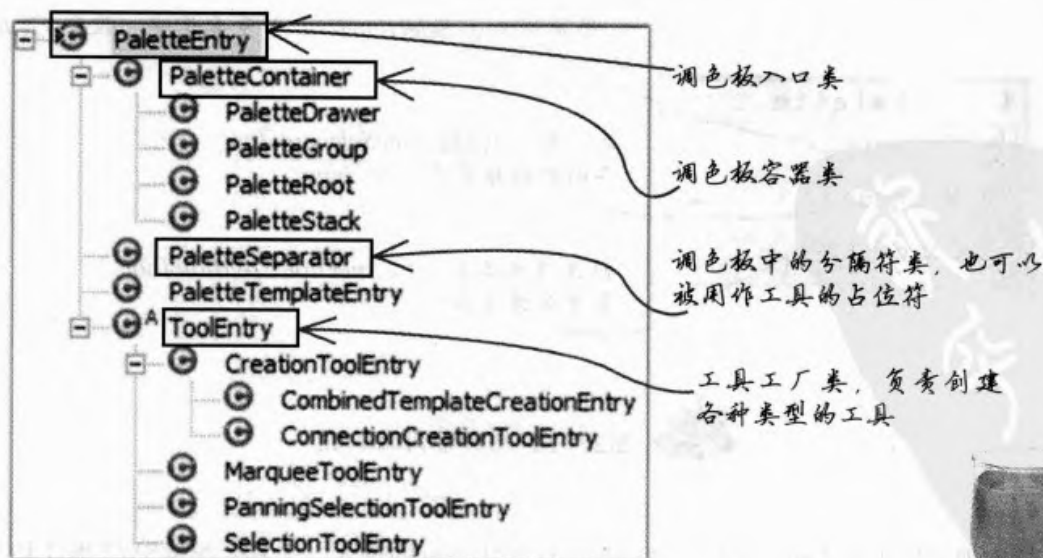


图22-25 调色板相关类的树形结构



除了PaletteRoot之外GEF中还有PaletteGroup和PaletteDrawer两个调色板容器。在构建调色板时建议读者同时使用这两个容器，它们都提供了PaletteEntry，但PaletteGroup不可以折叠，而PaletteDrawer可以被折叠。

继承自GraphicalEditorWithFlyoutPalette类的Editor可以实现带可浮动调色板样式的编辑器。下面代码中的SampleEditor扩展了GraphicalEditorWithFlyoutPalette并实现了getPaletteRoot()方法。

```
public class SampleEditor extends GraphicalEditorWithFlyoutPalette
implements ITabbedPropertySheetPageContributor{
    protected PaletteRoot getPaletteRoot() {
        if (this.paletteRoot == null) {
            this.paletteRoot = PaletteFactory.createPalette();
        }
        return this.paletteRoot;
    }
}
```

```
public class PaletteFactory {
    public static PaletteRoot createPalette() {
        PaletteRoot paletteRoot = new PaletteRoot();
        paletteRoot.addAll(createCategories(paletteRoot));
        return paletteRoot;
    }
}
```

在getPaletteRoot()中
从PaletteFactory创建
PaletteRoot并返回

工具是处理来自视图的大多数事件的类，当前活动工具对象由EditDomain负责维护。在大多数应用中常用的工具会有很多种，所以需要调色板在界面显示各种工具，并显示工具的描述信息。用户可以在调色板中单击选择不同的工具。图22-26描述了几种常用的工具和它们的用途。



图22-26 几种常用的工具

工具的实现类似于状态机，SWT 事件提供这些状态机的输入。工具会根据SWT事件和当前状态来执行各种行为 (Action)，这些行为包括以下几种。

★请求EditPart显示或隐藏反馈

- ★从EditPart中获取请求对应的Commands
- ★在命令堆栈中执行Commands
- ★更新鼠标光标

把工具设置到EditDomain中就等于将工具设置为当前活动工具，在任何时刻EditDomain中都只允许存在一个当前活动工具。如果使用了调色板，用户选中一个工具就等于将该工具设置为当前活动状态。下面着重介绍一下工具是如何工作的。

图22-27以选择工具——Selection Tool为例显示了工具的运行交互图。选择工具是GEF中最常用的工具也是大多数GEF应用中设置的默认工具。当使用选择工具拖动操作时，它可以从EditPart中获取一个DragTracker对象。从拖动操作鼠标键被按下开始到释放鼠标键产生的所有事件都会被发送给DragTracker，由DragTracker根据拖动产生的地点和方式处理事件。拖动操作对应的动作会因拖动位置的不同而不同，例如，拖动控制点意味着设定图形元素大小操作或移动连线端点操作，而拖动图形元素则是重新设定图形放置位置。

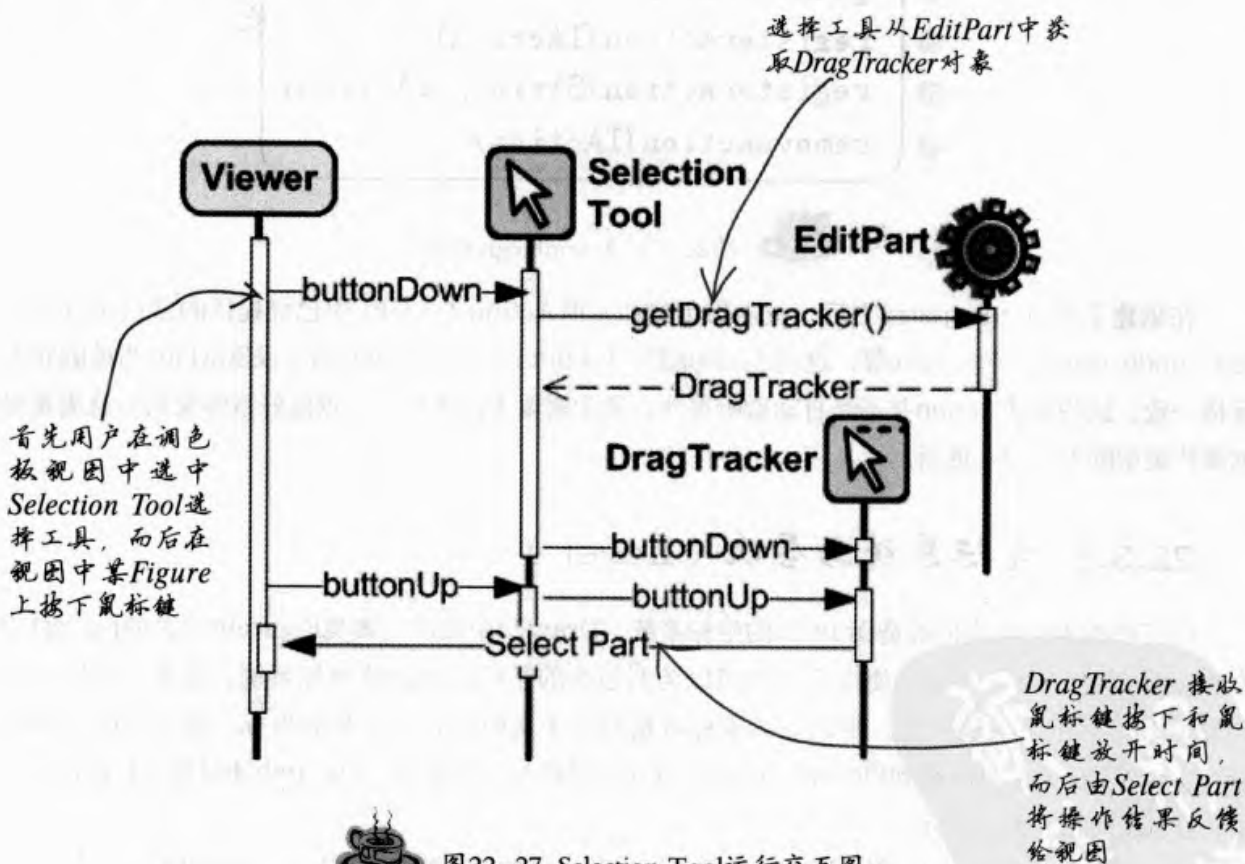


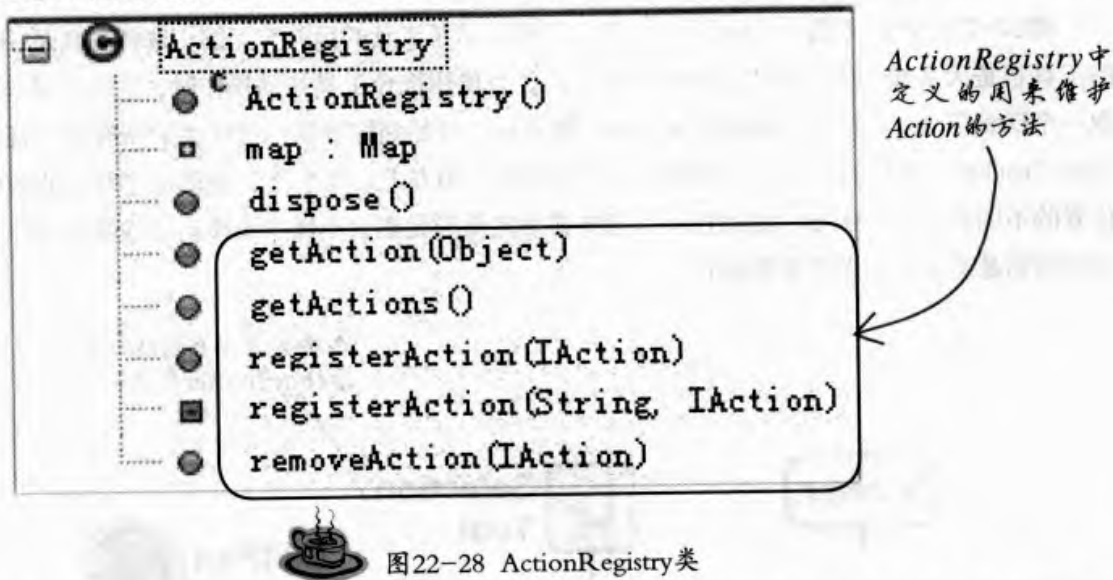
图22-27 Selection Tool运行交互图

实际上，选择工具的选择操作并不选中图形元素对应的EditPart，所有的鼠标单击都会被当成拖动操作处理。当选择工具接收到用户单击可选中EditPart产生的鼠标点下事件时，它就会申请一个Drag Tracker。此时，EditPart就会返回一个继承自SelectEditPartTracker的Tracker，它也会接收处理鼠标点下事件和其他的事件，直到鼠标键释放。Tracker也可以处理鼠标双击事件。

22.3.7 行为 (Action)

Action是Eclipse平台中菜单项, 工具栏按钮或context菜单项产生的动作请求在Workbench中的处理对象。GEF中提供了一系列的标准Action以及使用这些Action的框架结构。

GEF中的行为注册器类org.eclipse.gef.ui.actions.ActionRegistry是编辑器中Actions的容器类。ActionRegistry负责创建和维护Actions。图22-28列出了ActionRegistry类的结构。



在创建了ActionRegistry之后, 就能够创建和注册Action了。GEF中已经提供的常用Action有redo, undo, delete, print, save等, 这些Actions需要与Editor, CommandStack或EditPart当前的状态保持一致。这些默认Action并不是自动监听事件, 通常需要手动更新, 在创建好事件架构后就需要根据事件类型的不同加入更新。

22.3.8 坐标系统与层次 (Layer)

GEF中的坐标系统也就是Draw2d的坐标系统。Draw2d中图形元素类Figure中包含的protected方法useLocalCoordinates()使得Figure可以为其包含的子Figure设置坐标系统。如果一个Figure的父Figure使用了本地坐标, 那么它的坐标就是相对于父Figure左上角的坐标。通过Figure中的getClientArea()方法可以得到Figure所包含的子Figure能够显示的区域。Figure中还提供了转换Figure所在坐标的方法, 如下所示。

- ★translateToParent(): 将在Figure本地坐标内某点坐标值转换为其父Figure坐标值。
- ★translateFromParent(): 将Figure父坐标内某点坐标值转换为Figure本地坐标值。
- ★translateToRelative(): 将绝对坐标系统转换为相对Figure的相对坐标系统。
- ★translateToAbsolute(): 将相对Figure的相对坐标系统转换为绝对坐标系统。

注意: 锚点Anchor和定位器Locator的坐标都是绝对坐标。

在前面提到的RootEditPart实现中, 引入了层次的概念。层次的功能是管理控制EditPart中

Figure之间的层次关系。所有的Figure都处于主层次（Primary Layer）之中。表示连线的Figure处于连接层（Connection Layer）之中以确保连线画在所有Figure的最顶层。特殊的Figure，例如，拖放反馈和控制器（handler）位于特殊层（Special Layer）之中。特殊层位于Scalable Layer之上。图22-29以树的形式描述了几个主要层次的包含关系。

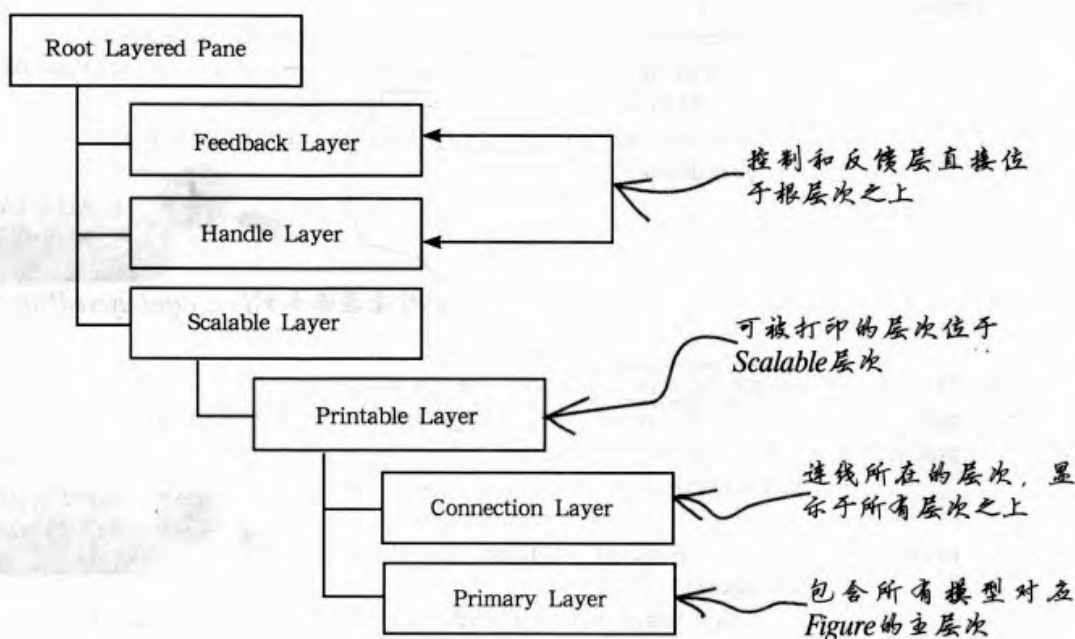


图22-29 层次关系树

层次是分组图形元素的容器，在层次中可以使用Z序控制图形元素的可见性。下面着重介绍一下层次的设置和使用。

在GEF的根EditPart类里面，如ScalableRootEditPart，FreeformGraphicalRootEditPart和ScalableFreeformRootEditPart都提供了允许他们的子类修改层次结构的方法。Root EditPart类中的Protected方法createLayers()和createPrintableLayers()负责设置层次，FreeformGraphicalRootEditPart中对这两个方法的实现如下所示。

```

protected void createLayers(LayeredPane layeredPane) {
    layeredPane.add(getPrintableLayers(), PRINTABLE_LAYERS);
    layeredPane.add(new FreeformLayer(), HANDLE_LAYER);
    layeredPane.add(new FeedbackLayer(), FEEDBACK_LAYER);
}
  
```

LayerPane 是一种用于维护层次的Figure，FreeformLayeredPane 则是专门用来维护FreeformLayer的LayerPane

```

protected LayeredPane createPrintableLayers() {
    FreeformLayeredPane layeredPane = new FreeformLayeredPane();
    layeredPane.add(new FreeformLayer(), PRIMARY_LAYER);
    layeredPane.add(new ConnectionLayer(), CONNECTION_LAYER);
    return layeredPane;
}
  
```


下面代码描述了向Editor中添加一个自定义的RootEditPart类——MyRootEditPart，并在类中重写createPrintableLayers()方法，创建一个渐变色填充的背景色层次(代码见工程HelloGEF，hellogef.editors.MyRootEditPart.java)。

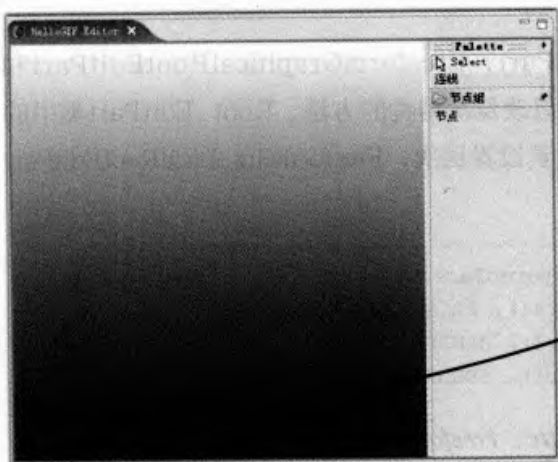
```
public class MyRootEditPart extends ScalableFreeformRootEditPart {
    protected LayeredPane createPrintableLayers() {
        FreeformLayeredPane layeredPane = new FreeformLayeredPane();
        layeredPane.add(new BackgroundLayer(), BackgroundLayer.BACKGROUND_LAYER);
        layeredPane.add(new FreeformLayer(), PRIMARY_LAYER);
        layeredPane.add(new ConnectionLayer(), CONNECTION_LAYER);
        return layeredPane;
    }
}
```

创建自定义的RootEditPart类 MyRootEditPart

把背景层添加到FreeformLayeredPane中

```
public class BackgroundLayer extends FreeformLayer {
    public static final String BACKGROUND_LAYER = "Background Layer";
    public BackgroundLayer() {
        setOpaque(true);
    }
    protected void paintFigure(Graphics graphics) {
        if(isOpaque()) {
            graphics.setForegroundColor(ColorConstants.white);
            graphics.setBackgroundColor(ColorConstants.blue);
            graphics.fillGradient(getBounds(), true);
        }
    }
}
```

创建扩展自FreeformLayer的层次BackgroundLayer



以蓝白渐变色填充层次

向GraphicalEditor中的GraphicalViewer视图中设置MyRootEditPart

```
public class DataMasterEditor extends GraphicalEditorWithFlyoutPalette
implements ITabbedPropertySheetPageContributor{
    protected void configureGraphicalViewer() {
        super.configureGraphicalViewer();
        getGraphicalViewer().setRootEditPart(new MyRootEditPart());
    }
}
```

GEF中包含的支持可测量（scaling）功能的EditPart有ScalableRootEditPart和ScalableFreeformRootEditPart，它们提供的方法中包含有创建可测量层次的方法，如下所示。

```
protected ScalableFreeformLayeredPane createScaledLayers();
```

通过继承RootEditPart类，可以控制层次的排列顺序，自定义层次的可打印和可测量特性。需要自定义层次的使用场合为编辑器提供一个背景层，可以被自由打开关闭的图形注释层等。

当在Editor中使用ScalableFreeformRootEditPart作为RootEditPart时，Editor可以向所有方向扩展编辑区域，因此，可以支持负坐标；而ScalableRootEditPart只能够支持正坐标。在使用ScalableFreeformRootEditPart创建GEF应用时，根模型对应的EditPart中使用的Figure必须是FreeformFigure类型的。

※ 注意 : ※

GEF中默认的层次组织是有规则的，如下所示。

1. 只有主画图和connection层才可以被打印。
2. Feedback层在Z序的最顶端，而后是handle层，最后是printable层。
3. 在可打印的层次中，connection层位于主画图之上。



22.4 创建GEF应用

通过以上介绍，读者应该已经对GEF的基本结构和概念有了初步的了解，本章就通过创建一个GEF图形编辑器——HelloGEF应用实例来详细讲解创建GEF应用的具体步骤。

HelloGEF实例是一个简单流程定义图形工具，能被用来描述工作流和数据流程图。可以通过属性框或直接单击节点图形元素修改图形主题，并通过连线工具有向连接两个节点。

典型的GEF应用程序一般都会会有一个图形编辑区域和一个调色板，它包含在一个Editor（例如GraphicalEditorWithPalette）里，可能还会包括一个大纲视图和一个属性页。

★物理上一个GEF应用的基本组成部分大致上包括以下几个。

- ★一组模型类
- ★一组EditPart
- ★一组EditPolicy
- ★一组修改模型的Command
- ★编辑器相关类

创建GEF应用的主要步骤和所需要完成的工作可以概括为如表22-2所示。



表22-2 创建GEF应用的主要步骤描述

主要步骤	需要完成的工作
添加模型	创建GEF应用的用户模型
添加视图	创建在用户界面上表现模型的图形视图

续表

添加控制器	创建GEF架构中的协调控制中心——EditPart
创建编辑器	创建Eclipse平台编辑器
添加缩放、直接编辑等附属功能	向GEF应用添加便于用户在图形界面编辑模型的附属功能，常用的附属功能包括缩放和直接编辑功能等。
添加编辑策略	向EditPart中添加编辑策略（EditPolicy），这一步骤一般在创建EditPart时一同完成
添加属性页	添加辅助编辑模型属性的属性页
添加调色板和调色板中工具	添加图形编辑工具和其容器——调色板

创建GEF应用的步骤顺序大多是先设计模型和视图，然后设计控制器，并逐渐增加各种编辑行为。这只是一般步骤，实际应用中可灵活变化，如可先实现编辑器，并将要编辑的各组件放置在Palette面板上。这样做可以对要做的应用有一个把握，然后再设计模型、控制器并逐渐增加各种编辑行为，视图则在控制器的实现过程中逐步实现。如果需要连接线连接图形节点，如ETL流程编辑器、工作流编辑器等流程图编辑工具，就需要添加连接线Connections了。这就需要向GEF应用中添加连接线的模型、视图、控制器。创建一个GEF应用的一般步骤大致可以描述为如图22-30所示。

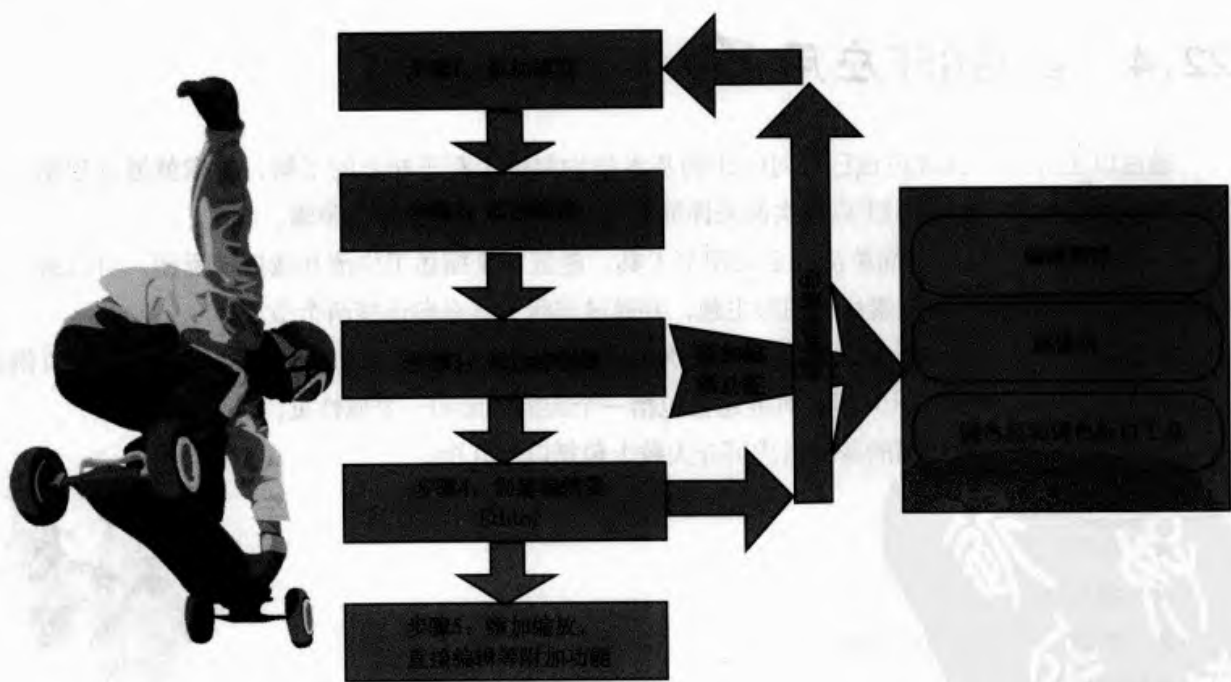
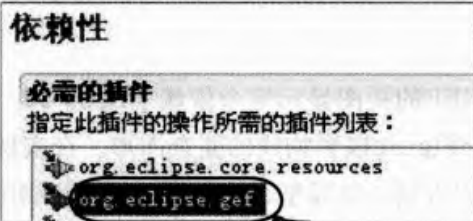


图22-30 构建GEF应用的步骤

首先创建一个Plugin工程，工程名称为HelloGEF。然后在插件依赖中添加Editor扩展点相关依赖插件和org.eclipse.gef插件，如图22-31所示。这一过程已经在本书的其他章节中做了详细介绍，这里就不再重述了。



把GEF包添加到HelloGEF工程的插件依赖中



图22-31 添加GEF依赖

22.4.1 添加模型

模型是MVC架构中最基础的部分，也是GEF应用需要编辑的内容，所以在创建GEF应用之前必须构建好用户模型。模型不需要关心任何关于视图和控制器的信息，因此为了能让控制器获取模型的变化，通常把控制器作为事件监听者注册在模型中，当模型发生变化时，就向控制器发送消息，后者负责通知并更新各个视图。下面是模型的抽象类AbstractModel的代码描述，它需要实现Cloneable，Serializable和IPropertySource接口。Cloneable和Serializable接口保证了AbstractModel可以被克隆和被持久化，IPropertySource则使得AbstractModel可以作为标准属性框的属性来源。

典型的模型包括节点对应的大小、位置等信息，对于与其他具有连接关系的模型，还要维护连接信息。此外，模型对象通常包含PropertyChangeSupport类型的成员变量，用来维护监听器成员。模型的代码描述如图22-32所示(代码见工程HelloGEF，hellogef.model.NodeModel.java)。

```
public class NodeModel extends AbstractModel {
    protected IPropertyDescriptor[] descriptors = new IPropertyDescriptor[] {
        new ComboBoxPropertyDescriptor (PROP_VISIBLE, "是否可见", new
String[] { "可见", "不可见" } );
        new TextPropertyDescriptor (PROP_NAME, "名称"),
    };
    public NodeModel(){
        this.name = "节点";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        if (this.name.equals(name)) {
            return;
        }
        this.name = name;
        firePropertyChange (PROP_NAME, null, name);
    }
}
```

定义以下拉列表框编辑“名称”属性

定义以文本框编辑“名称”属性

发送事件通知EditPart 模型名称属性已修改

属性	值
名称	节点
是否可见	可见



图22-32 创建模型

22.4.2 添加视图

视图是模型在图形编辑器上对应显示的图形，GEF的视图显示完全依赖于Draw2d，因此，模型的图形元素一般采用单独的Figure简单图形或多个Figure嵌套构成的复杂图形。在实现过程中，Figure可以由EditPart控制图形绘制和显示方式也可以直接关联模型，按照模型属性绘制图形。下面是继承自抽象模型类AbstractModel的节点模型NodeModel对应图形元素NodeFigure的代码实现(代码见工程HelloGEF，hellogef.view.NodeFigure.java)。

```
public class NodeFigure extends Shape {
    private String name;
    private Label label;
    public NodeFigure() {
        this.label = new Label();
        this.add(label);
    }
    public String getText() {
        return this.label.getText();
    }
    public Rectangle getTextBounds() {
        return this.label.getTextBounds();
    }
}
```

用于显示节点名称的Figure，它是Draw2d中提供的org.eclipse.draw2d.Label 标签控件

```
public void setName(String name) {
    this.name = name;
    this.label.setText(name);
    this.repaint();
}
public void setBounds(Rectangle rect) {
    super.setBounds(rect);
    this.label.setBounds(rect);
}
```

设置Label标签控件的名称

设置Label标签控件的位置、大小

```
protected void fillShape(Graphics graphics) {
    graphics.pushState();
    Rectangle bound = this.getBounds().getCopy();
    graphics.setBackgroundColor(ColorConstants.lightGray);
    graphics.fillRoundRectangle(bound, 30, 30);
    graphics.popState();
}
```

先把背景色设为浅灰色，而后以背景色填充Figure区域内的圆角矩形

```
protected void outlineShape(Graphics graphics) {
    Rectangle r = getBounds().getCopy();
    r.x = r.x + lineWidth / 2;
    r.y = r.y + lineWidth / 2;
    r.width = r.width - Math.max(1, lineWidth);
    r.height = r.height - Math.max(1, lineWidth);
    graphics.drawRoundRectangle(r, 30, 30);
}
```

设置Figure区域宽高，而后以前景色绘制圆角矩形



22.4.3 创建控制中心——EditPart

到目前为止，已经介绍了创建GEF编辑器所要用的模型和视图，现在可以向编辑器中添加控制中心EditPart并映射模型元素了。创建模型映射的第一步是构建对应于模型的EditPart。最简单的构建EditPart方式是创建与模型一一对应的EditPart结构，参照模型之间的结构构建EditPart之间的关系。有些时候这种一一对应的EditPart并不能满足需要，还要采用其他的EditPart创建策略。但不管采用什么方式创建，都需要一个作为入口的contentEditPart。每一个EditPartViewer只允许包含一个contentEditPart。contentEditPart与RootEditPart并不相干，但某些RootEditPart（例如ScalableFreeformRootEditPart）中有可能为contentEditPart的Figure定义一些限制信息。

某些情况下EditPart还需要实现NodeEditPart接口，通过实现这个接口，EditPart可以监听模型属性修改事件。NodeEditPart接口是默认的ConnectionEditPart实现中用来为连线图元Figure类org.eclipse.draw2d.Connection获取连接锚点ConnectionAnchors。在执行过程中依赖于此接口的类有如下几个。

★org.eclipse.gef.editPart.AbstractConnectionEditPart：在执行refresh()方法时，AbstractConnectionEditPart 使用NodeEditPart接口为它对应的Figure获取ConnectionAnchors。

★org.eclipse.gef.editPolicy.GraphicalNodeEditPolicy：在新建连线时，并不存在ConnectionEditPart，因此作为连接源的起点负责利用NodeEditPart接口显示反馈。

★org.eclipse.gef.editPolicy.ConnectionEndpointEditPolicy：在改变连线连接位置的时候，ConnectionEndpointEditPolicy利用NodeEditPart接口获取合适的连接目的ConnectionAnchors。

实现了PropertyChangeListener接口的EditPart可以接收到来自于模型的属性修改请求。当模型属性发生改变后，模型会发送属性修改事件，并附上修改属性的键值，然后GEF会将事件传递给模型对应的EditPart的propertyChange(PropertyChangeEvent evt)方法。这个方法负责判断模型修改事件，并执行相应的视图更新。下面代码详细描述了这一过程(代码见工程HelloGEF, hellogef.model.AbstractModel.java)。

```
public abstract class AbstractModel implements Cloneable, Serializable,
```

```
    IPropertySource {
```

```
        public void setName(String name) {
```

```
            if (this.name.equals(name)) {
```

```
                return;
```

```
            }
```

```
            this.name = name;
```

```
            firePropertyChange(PROP_NAME, null, name);
```

```
        }
```

```
IPropertySource
+ getEditableValue()
+ getPropertyDescriptors()
+ getPropertyValue(Object)
+ isPropertySet(Object)
+ resetPropertyValue(Object)
+ setPropertyValue(Object, Object)
```

当模型修改时产生属性修改事件
PropertyChangeEvent

```
protected void firePropertyChange(String prop, Object old, Object newValue) {
```

```
    listeners.firePropertyChange(prop, old, newValue);
```

```
}
```


实现PropertyChangeListener接口中定义的propertyChange(PropertyChangeEvent evt)方法监听模型产生的属性修改事件PropertyChangeEvent

```
public class NodePart extends AbstractGraphicalEditPart
implements PropertyChangeListener, NodeEditPart {
    public void propertyChange(PropertyChangeEvent evt) {
        if (evt.getPropertyName().equals(NodeModel.PROP_LOCATION))
            refreshVisuals();
        else if (evt.getPropertyName().equals(NodeModel.PROP_NAME))
            refreshVisuals();
        else if (evt.getPropertyName().equals(NodeModel.PROP_INPUTS))
            refreshTargetConnections();
        else if (evt.getPropertyName().equals(NodeModel.PROP_OUTPUTS))
            refreshSourceConnections();
    }
}
```

① PropertyChangeListener
② propertyChange(PropertyChangeEvent)

下面是对应于NodeModel模型和NodeFigure视图的EditPart-NodePart类的完整代码实现(代码见工程HelloGEF, hellogef.control.NodePart.java)。

```
public class NodePart extends AbstractGraphicalEditPart
implements PropertyChangeListener, NodeEditPart {
    protected DirectEditManager manager;
    public void performRequest(Request req) {
        if (req.getType().equals(RequestConstants.REQ_DIRECT_EDIT)) {
            if (manager == null) {
                NodeFigure figure = (NodeFigure) getFigure();
                manager = new NodeDirectEditManager(this, TextCellEditor.class, new
                NodeCellEditorLocator(figure));
            }
            manager.show();
        }
    }
    public void propertyChange(PropertyChangeEvent evt) {
        if (evt.getPropertyName().equals(NodeModel.PROP_LOCATION))
            refreshVisuals();
        else if (evt.getPropertyName().equals(NodeModel.PROP_NAME))
            refreshVisuals();
        else if (evt.getPropertyName().equals(NodeModel.PROP_INPUTS))
            refreshTargetConnections();
        else if (evt.getPropertyName().equals(NodeModel.PROP_OUTPUTS))
            refreshSourceConnections();
    }
    protected IFigure createFigure() {
        return new NodeFigure();
    }
    protected void createEditPolicy() {
        installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new NodeDirectEditPolicy());
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new NodeEditPolicy());
    }
}
```

处理指定类型的请求
预定义的直接编辑请求
处理属性修改事件, 根据事件中被修改属性的类型更新视图
创建视图 Figure
创建所需角色对应的各种EditPolicy

```

installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new NodeGraphicalNodeEditPolicy());
}

public void activate() {
    if (isActive()) {
        return;
    }
    super.activate();
    ((NodeModel) getModel()).addPropertyChangeListener(this);
}

public void deactivate() {
    if (!isActive()) {
        return;
    }
    super.deactivate();
    ((NodeModel) getModel()).removePropertyChangeListener(this);
}

protected void refreshVisuals() {
    NodeModel node = (NodeModel) getModel();
    Point loc = node.getLocation();
    Dimension size = new Dimension(150, 40);
    Rectangle rectangle = new Rectangle(loc, size);
    ((NodeFigure) this.getFigure()).setName(((NodeModel) this.getModel()).
    getName());
    ((GraphicalEditPart) getParent()).setLayoutConstraint(this, getFigure(),
    rectangle);
}

public ConnectionAnchor getSourceConnectionAnchor(ConnectionEditPart connection) {
    return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getSourceConnectionAnchor(Request request) {
    return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(ConnectionEditPart connection) {
    return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(Request request) {
    return new ChopboxAnchor(getFigure());
}

protected List getModelSourceConnections() {
    return ((NodeModel) this.getModel()).getOutgoingConnections();
}

protected List getModelTargetConnections() {
    return ((NodeModel) this.getModel()).getIncomingConnections();
}
}

```

名用，禁用 EditPart

根据对应模型的属性值刷新视图

返回连接到该 EditPart 连接线的起点和终点锚点

前面提到过，EditPart 的创建工作通常情况下由一个工厂类来完成，这个工厂类需要实现 EditPartFactory 接口。在接口中只定义了一个方法，在这个方法中，需要实现的是为指定模型元素创建与其对应的 EditPart。在创建完 EditPart 之后，还需要调用 EditPart 中定义的 setModel() 方法把

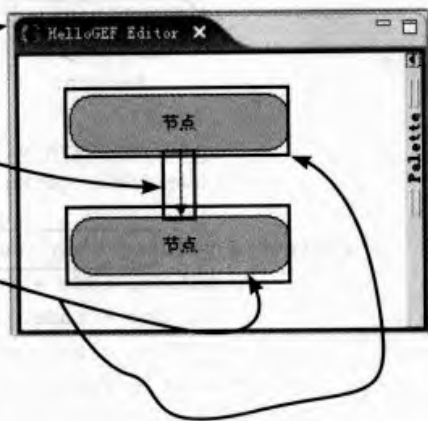
模型元素对象设置到已创建的EditPart之中。图22-33是PartFactory 类对EditPartFactory的实现代码描述，其中定义了模型与控制器EditPart的对应关系(代码见工程HelloGEF，hellogef.control.PartFactory.java)。

```
public class PartFactory implements EditPartFactory {
    public EditPart createEditPart(EditPart context, Object model) {
        EditPart part = null;
        if (model instanceof Diagram)
            part = new DiagramPart();
        else if (model instanceof ConnectionModel)
            part = new ConnectionPart();
        else
            part = new NodePart();
        part.setModel(model);
        return part;
    }
}
```

EditPartFactory中定义的
各个模型对应EditPart 在
图形上的映射



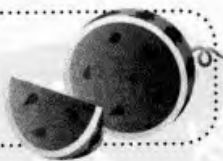
图22-33 PartFactory



创建完EditPart后，它还需要把自己注册到注册器里面去。在EditPartViewer中维护有EditPart注册器，它负责所有模型对应的EditPart的注册管理。EditPart的默认实现中已经包含了这部分功能，并把其对应的模型元素作为键值。用EditPart注册器来维护EditPart和模型之间的对应关系是一种相对安全的实现方案。一种需要EditPart注册器维护EditPart模型关系的情况是采用全局监听器模型，所有模型元素共用此监听器发送事件，监听器负责接收所有模型事件对象，它需要利用EditPart注册器寻找模型元素对应的EditPart。

※ 注意: ※

对于含有子元素的EditPart，必须覆盖getModelChildren()方法返回子对象列表。



22.4.4 创建编辑器——Editor

编辑器——Editor是GEF应用的主要组成部分，也是用户进行模型对应的图形元素编辑的区域。在Eclipse插件项目中添加自定义Editor需要扩展editor扩展点，这个过程已经在本书中其他部分做了详细介绍，在这里就不再赘述了。默认情况下editor类直接扩展自org.eclipse.ui.part.EditorPart，它是Eclipse平台下编辑器editor的基础类，负责接收、处理、保存input和创建，设置viewer。

下面是扩展自GraphicalEditorWithFlyoutPalette的编辑器HelloEditor的代码实现(代码见工程HelloGEF，hellogef.editors.HelloEditor.java)。

```

public class HelloEditor extends GraphicalEditorWithFlyoutPalette {
    private Diagram diagram = new Diagram();
    private PaletteRoot paletteRoot;
    public Diagram getDiagram() {
        return this.diagram;
    }
    public HelloEditor() {
        setEditDomain(new DefaultEditDomain(this));
    }
    protected void configureGraphicalViewer() {
        super.configureGraphicalViewer();
        getGraphicalViewer().setRootEditPart(new MyRootEditPart());
        getGraphicalViewer().setEditPartFactory(new PartFactory());
    }
    protected void initializeGraphicalViewer() {
        getGraphicalViewer().setContents(this.diagram);
        getGraphicalViewer().addDropTargetListener(new DiagramTemplateTransferDropTarget-
tListener(getGraphicalViewer()));
    }
    public void doSave(IProgressMonitor monitor) {
    }
    public void doSaveAs() {
    }
    public boolean isDirty() {
        return getCommandStack().isDirty();
    }
    public boolean isSaveAsAllowed() {
        return true;
    }
    protected void setInput(IEditorInput input) {
        super.setInput(input);
        diagram = new Diagram();
    }
    public Object getAdapter(Class type) {
        if (type == IContentOutlinePage.class)
            return new OutlinePage();
        return super.getAdapter(type);
    }
    protected PaletteRoot getPaletteRoot() {
        if (this.paletteRoot == null) {
            this.paletteRoot = PaletteFactory.createPalette();
        }
        return this.paletteRoot;
    }
}

```

在Editor 类的构造函数中设置EditDomain

完成保存编辑结果操作

完成编辑结果另存为操作

返回CommandStack 中是否有保存后执行的Command 的布尔值

设置是否允许另存为操作

为Editor 设置输入，此处当前代码是新建一个Diagram 模型，读者可以在这里加入从文件读取模型操作，实现编辑保存在文件的模型的功能

在此设置各种自定义控件，比如添加自定义大纲视图，自定义属性视图等

在此添加创建调色板PaletteRoot 代码

1. 向Editor中加入键盘控制

GEF架构中以KeyHandler处理键盘事件。默认情况下，GEF中的GraphicalViewer并不响应键盘

事件，在需要的时候得自己加上去。在GEF中加入键盘事件处理并不是难事，因为GEF中已经提供了KeyHandler的默认实现org.eclipse.gef.ui.parts.GraphicalViewerKeyHandler。示例代码如下所示。

```
private GraphicalViewer createGraphicalViewer(Composite parent)
{
    //创建GraphicalViewer视图
    GraphicalViewer viewer = new ScrollingGraphicalViewer();
    viewer.createControl(parent);
    //配置视图
    viewer.getControl().setBackground(parent.getBackground());
    viewer.setRootEditPart(new ScalableFreeformRootEditPart());

    viewer.setKeyHandler(new GraphicalViewerKeyHandler(viewer));

    //将创建的视图添加到EditDomain
    getEditDomain().addViewer(viewer);
    //将视图设为Eclipse平台的选择提供者 - Selection provider
    getSite().setSelectionProvider(viewer);
    //向视图中添加输入初始化视图
    viewer.setEditPartFactory(getEditPartFactory());
    viewer.setContents(getContent());
    return viewer;
}
```

向视图添加KeyHandler



在对键盘事件附加actions的时候，并不需要复写GraphicalViewerKeyHandler，只需要创建并配置自己的KeyHandler实例，而后把这个实例设为添加到GraphicalViewer中的GraphicalViewerKeyHandler实例的parent就可以了。

2. 创建EditDomain

在创建了Editor编辑器以后需要一个EditDomain来把editor，viewers和tools逻辑上绑定在一起。EditDomain中提供了命令堆栈CommandStack。CommandStack负责管理应用中所有已经执行的Commands。CommandStack是实现Redo/Undo功能的基础，并且可以用来判断editor是否已经被修改以确定是否需要保存。通常情况下一个editor包含一个EditDomain，但有些多页面的编辑器中，多个editor也可以共享一个EditDomain。

在实现EditDomain时，可以直接使用GEF中提供的默认实现DefaultEditDomain类。下面是GEF中DefaultEditDomain类的实现代码。

```
public class DefaultEditDomain
    extends EditDomain
{
    private IEditorPart editorPart;
    public DefaultEditDomain(IEditorPart editorPart) {
        setEditorPart(editorPart);
    }
}
```

```

public IEditorPart getEditorPart() {
    return editorPart;
}

protected void setEditorPart(IEditorPart editorPart) {
    this.editorPart = editorPart;
}
}

```

下面列出了HelloEditor中创建DefaultEditDomain，并在HelloEditor的构造函数中把它设置到Editor的实现代码。

```

public class HelloEditor extends GraphicalEditorWithFlyoutPalette {
    public HelloEditor() {
        setEditDomain(new DefaultEditDomain(this));
    }
}

```

3. 可替换 (Adaptable) 功能

Eclipse平台中提供的一个重要的功能就是其组件的可替换性。GEF也继承了Eclipse平台的这一特性。GEF中的组件通过实现org.eclipse.core.runtime.IAdaptable接口保证了实现此接口类的可替换的机制，可以在IAdaptable中定义的getAdapter()加入其他的GEF组件实现替换。下面列出了向HelloEditor类的getAdapter()方法中添加自定义的GraphicalViewer，CommandStack，EditDomain和OutLinePage的代码描述。

```

public class HelloEditor extends GraphicalEditorWithFlyoutPalette {
    public Object getAdapter(Class adapter)
    {
        if (adapter == GraphicalViewer.class || adapter == EditPartViewer.class)
            return getGraphicalViewer();

        else if (adapter == CommandStack.class)
            return getCommandStack();

        else if (adapter == EditDomain.class)
            return getEditDomain();

        else if (type == IContentOutlinePage.class)
            return new OutlinePage();

        return super.getAdapter(adapter);
    }
}

```

以自定义的GraphicalViewer，CommandStack，EditDomain和OutLinePage替换默认实现

22.4.5 添加调色板 (palette)

调色板本身就是一个视图，它扩展自ScrollingGraphicalViewer类。向编辑器中添加调色板类似于添加视图的过程。首先需要创建一个PaletteViewer，代码如下所示。


```
private PaletteViewer createPaletteViewer(Composite parent)
{
    PaletteViewer viewer = new PaletteViewer();
    viewer.createControl(parent);
    getEditDomain().setPaletteViewer(viewer);
    getEditDomain().setPaletteRoot(getPaletteRoot());
    return viewer;
}
```

将新建的PaletteViewer设置到EditDomain中，一个EditDomain只允许设置一个PaletteViewer
向EditDomain添加PaletteRoot

```
protected PaletteRoot getPaletteRoot() {
    if (this.paletteRoot == null) {
        this.paletteRoot = PaletteFactory.createPalette();
    }
    return this.paletteRoot;
}
```

```
public class PaletteFactory {
    public static PaletteRoot createPalette() {
        PaletteRoot paletteRoot = new PaletteRoot();
        paletteRoot.addAll(createCategories(paletteRoot));
        return paletteRoot;
    }
    private static List createCategories(PaletteRoot root) {
        List categories = new ArrayList();
        categories.add(createControlGroup(root));
        categories.add(createComponentsDrawer());
        return categories;
    }
    private static PaletteContainer createControlGroup(PaletteRoot root) {
        PaletteGroup controlGroup = new PaletteGroup("control");
        List entries = new ArrayList();
        ToolEntry tool = new SelectionToolEntry();
        entries.add(tool);
        root.setDefaultEntry(tool);
        tool = new ConnectionCreationToolEntry("连线", "创建连接节点的连线", null,
        null, null);
        entries.add(tool);
        controlGroup.addAll(entries);
        return controlGroup;
    }
    private static PaletteContainer createComponentsDrawer() {
        PaletteDrawer drawer = new PaletteDrawer("节点组");
        List entries = new ArrayList();
        ToolEntry tool = new CombinedTemplateCreationEntry("节点", "创建一个节点",
        NodeModel.class, new SimpleFactory(
        NodeModel.class), null, null);
        entries.add(tool);
        drawer.addAll(entries);
    }
}
```



在控制分组中分别添加默认的选择工具和连线工具

创建工具分组

向工具分组中添加节点工具


```
return drawer;
```

```
}
```

```
}
```

创建完调色板视图之后，还需要把视图加入editor的composite中。使用SWT中的SashForm可以使调色板可调整。如果需要调色板可以被用户按需求定制，就需要在调色板中加入定制工具——Palette Customizer。

PaletteRoot是调色板模型的根，实现调色板需要向EditDomain添加自定义的PaletteRoot在HelloEditor编辑器中，添加调色板PaletteRoot是通过实现GraphicalEditorWithFlyoutPalette中定义的getPaletteRoot()方法完成的，而PaletteRoot的创建则在PaletteFactory类中完成。



注意：如果编辑器是直接扩展GraphicalEditorWithPalette或GraphicalEditorWithFlyoutPalette实现的，就不需要再添加PaletteViewer了。



22.4.6 添加连接线——Connections

连线是在图形上表示连接两个Figure图形元素的连线，它的控制器ConnectionEditPart则表现为连接两个EditPart。ConnectionEditPart只是包含源端和目的端EditPart的GraphicalEditPart。添加连线首先要创建连线模型和模型对应的控制器ConnectionEditPart。连线控制器需要实现ConnectionEditPart接口。

创建连接线模型与创建节点模型相同，连接线模型中一般需要记录连接的起点和终点节点模型。如果连线有属性，它也可以实现IPropertySource接口，在属性界面显示，编辑连线属性。

下面是ConnectionPart连线控制器的实现代码(代码见工程HelloGEF, hellogef.control.ConnectionPart.java)，ConnectionPart直接扩展自GEF中提供的默认连接控制器实现——抽象类AbstractConnectionEditPart。

```
public class ConnectionPart extends AbstractConnectionEditPart {
    protected IFigure createFigure() {
        PolylineConnection conn = new PolylineConnection();
        conn.setTargetDecoration(new PolygonDecoration());
        conn.setConnectionRouter(new BendpointConnectionRouter());
        return conn;
    }
    protected void createEditPolicy() {
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new ConnectionEditPolicy());
        installEditPolicy(EditPolicy.CONNECTION_ENDPOINTS_ROLE, new ConnectionEndpointEditPolicy());
    }
    protected void refreshVisuals() {
    }
    public void setSelected(int value) {
        super.setSelected(value);
        if (value != EditPart.SELECTED_NONE)
            ((PolylineConnection) getFigure()).setLineWidth(2);
    }
}
```

连接控制器的默认实现类

添加允许带有折点的连线Figure样式

设置PolylineConnection的端点标识

设置连线路由算法，这里用的是GEF中提供的默认路由实现BendpointConnectionRouter


```

else
    ((PolylineConnection) getFigure()).setLineWidth(1);
}
}

public class PartFactory implements EditPartFactory {
    public EditPart createEditPart(EditPart context, Object model) {
        EditPart part = null;
        if (model instanceof Diagram)
            part = new DiagramPart();
        else if (model instanceof ConnectionModel)
            part = new ConnectionPart();
        else
            part = new NodePart();
        part.setModel(model);
        return part;
    }
}

public class ConnectionModel extends AbstractModel {
    private NodeModel source;
    private NodeModel target;
    public ConnectionModel(NodeModel source, NodeModel target) {
        super();
        this.source = source;
        this.target = target;
        source.addOutput(this);
        target.addInput(this);
        this.name = "连线";
    }
}

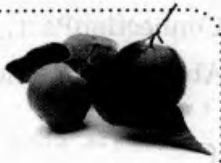
```

*ConnectionModel*继承自*AbstractModel*抽象模型，其中定义了需要在属性视图显示的名称属性，因此在图形上单击连线属性页中会显示连接线的名称属性

在构造函数中初始化连线的名称属性，用户可以在属性视图中编辑该属性

※ 注意 : ※

连线并不是直接连接到其他图形元素，位于他们之间的是连线源，目的 *EditPart* 所提供的连接锚点 (*ConnectionAnchors*)。GEF 建议在创建可以被连线连接的 *GraphicalEditPart* 时实现 *NodeEditPart* 接口，这也是创建模型对应图形结构最常用的方式。



GEF 中的连线是使用 Draw2d 中的 *PolylineConnection* 类画出来的，在实际使用的时候往往需要在连线上添加一些标识或装饰 (Decoration)，*PolylineConnection* 类中提供了向连线中添加标识的支持。

通过设置 *PolylineConnection* 连线标识类，如 *RotatableDecoration*，就能够在连线起点、终点，或者是两个端点添加相应的图形标识。*RotatableDecoration* 接口定义了允许自己围绕指定坐标旋转的功能，这就使得在移动连线连接的 Figure 移动时，能够改变自身位置与连线位置保持一致。图 22-34 和图 22-35 分别显示了不同用途的连线标识图形样式。

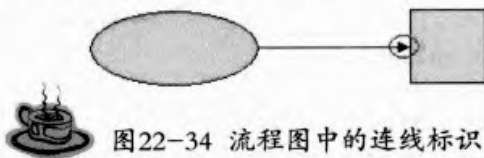


图 22-34 流程图中的连线标识图形样式

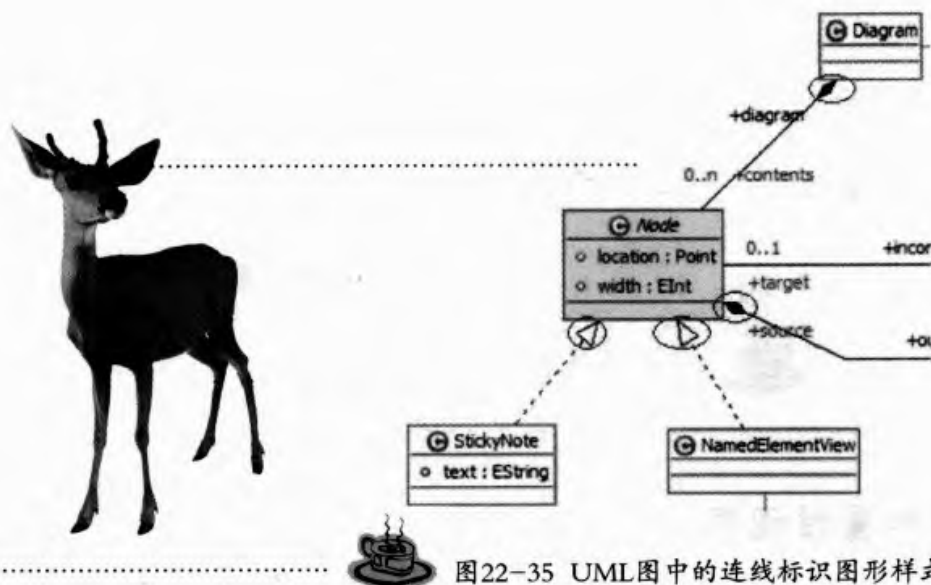


图22-35 UML图中的连线标识图形样式

最常用的连线标识就是箭头了，Draw2d中包含有两种RotatableDecoration的实现，分别是RotatablePolyline和RotatablePolygon。这两个类的构造函数都能够创建箭头标识。通过调用PolylineConnection中的setSourceDecoration()和setTargetDecoration()方法可以向PolylineConnection连线设置起始端点的标识。

在需要向连线添加自定义的标识的情况下，可以调用标识类中的setTemplate()方法，并传入一个包含所有需要添加标识的坐标的列表。通过调用setScale()也可以控制标识的大小。

PolyLineConnection使用DelegatingLayout布局管理器，这就意味着它的所有子Figure都需要提供继承自Locator的constraint。当用setSourceDecoration()和setTargetDecoration()向连线添加标识的时候，这些方法就会自动创建ArrowHeadLocator并把它设置成这个标识的constraint。箭头的定位器(Locator)会保证标识图标准确地连接到连线的末端位置。

在有些应用中，有可能需要向PolylineConnection连线上添加其他的一些Figure图形元素，如添加一些包含注释的标签。由于PolylineConnection类中有一个DelegatingLayout类型的布局管理器，如果需要在连线上添加注释标签，只需要创建标签Figure并作为子图元加入到PolylineConnection中就可以了；然后设置包含标签Figure位置信息的constraint。图22-36描述了向连线中点位置添加标签的过程。带中点标签描述的连线如图22-37所示。

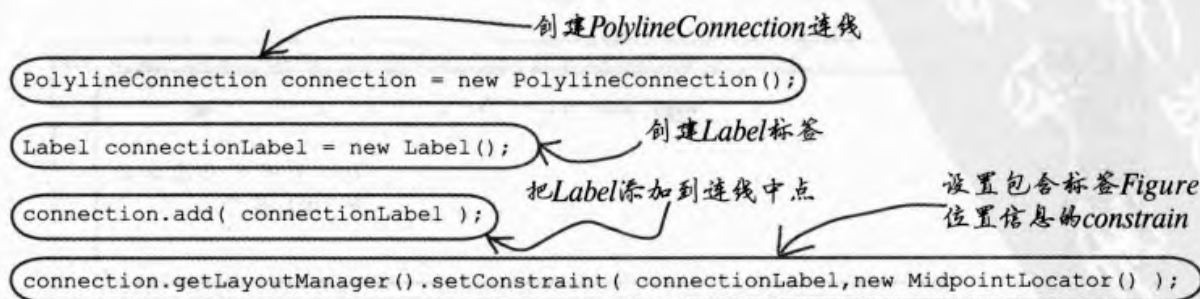


图22-36 在连线中点添加标签的过程

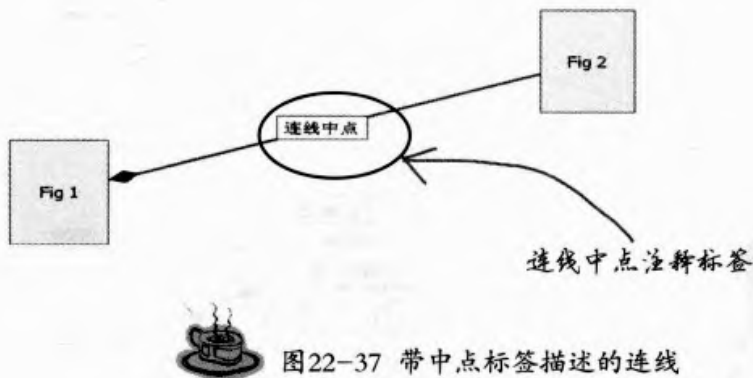


图22-37 带中点标签描述的连线

22.4.7 添加属性视图

属性视图主要是提供在图形界面编辑模型元素属性的功能。GEF框架中提供了监控标准属性视图中属性值的修改并创建相应的Commands 加入CommandStack中执行的机制，这么做不但可以实现可视化模型属性编辑功能，还可以实现Redo/Undo属性视图中的编辑动作。在这个过程中EditPart负责为属性视图提供属性来源模型，也就是实现了IPropertySource 接口的模型。

图22-38中显示了标准属性视图的样式，标准属性视图提供了一个属性表格控件，它提供了属性名、属性值两列，不但可以以简单列表形式显示属性，而且可以为属性创建分组显示。

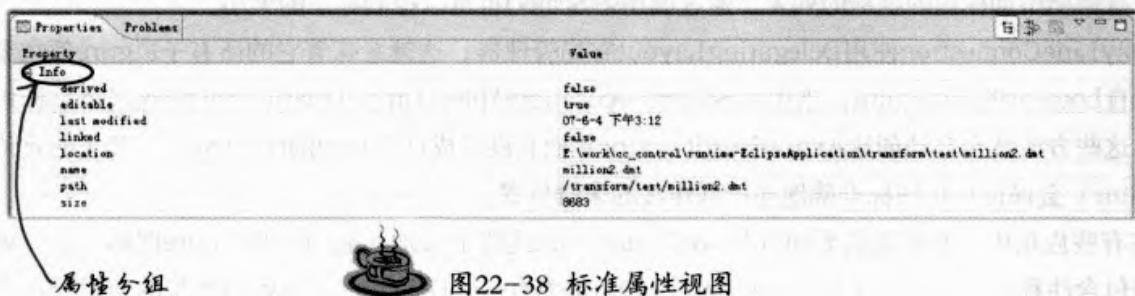
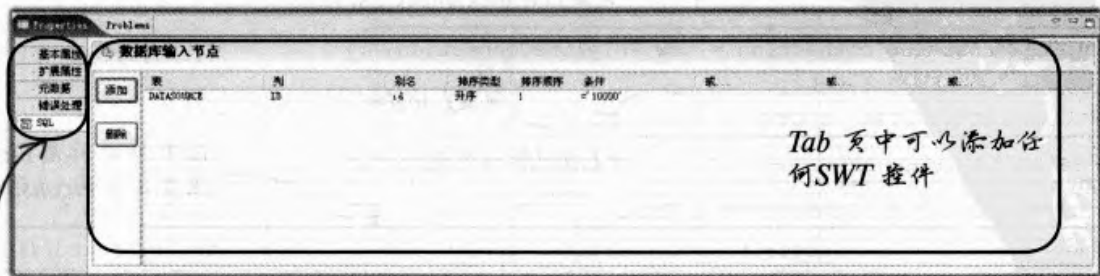


图22-38 标准属性视图

图22-39属性风格是Eclipse 3.2 M5之后的版本提供的标签分页格式的属性视图，这个视图中允许在标签页面中添加任何SWT控件构建各种形式的属性编辑页面。在实现过程中需要使用Eclipse平台中提供的org.eclipse.ui.views.properties.tabbed插件。



Tab属性页选择框

图22-39 标签分页属性视图

标签分页格式的属性视图的标签页面中还可以同时使用标准属性视图，两种风格的属性视图结合使用。图22-40为包含标准属性视图的标签分页属性视图。

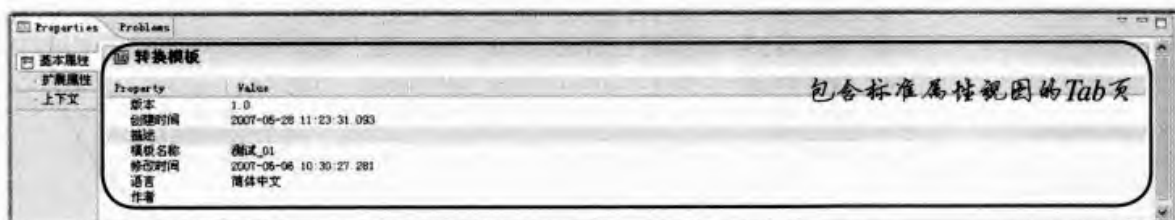


图22-40 包含标准属性视图的标签分页属性视图

GEF应用主要功能就是提供创建以图形的方式展现，编辑模型和模型中属性的工具的框架，需要在编辑器中为用户提供需要的图形布局方式。GEF框架中可以使用Draw2d中提供的所有布局管理器。一些布局需要使用坐标，大小之类的参数，这些参数往往维护在与图形元素对应的模型元素中。在保存模型时，可以将这些图形相关参数一并保存以便于再次打开图形时能够恢复原编辑状态各个图形元素的状态。这些参数可以被存储为模型元素的属性或是描述信息，也可以与模型分开存储。

如果编辑器中一个模型元素被编辑了或某些状态被改变了，那么，必须保证与它对应的图形元素状态也要被更新，这首先需要控制器Controller与模型元素之间产生交互。在实现的时候如果让模型直接与其对应的控制器之间直接关联并不是理想的方案。通常的做法是在改变模型属性的同时，创建某种特定类型的事件对象并发送（fire）出去。控制器可以注册监听这个模型元素或事件管理器监听指定类型的事件并完成相应的处理动作。在EditPart中完成这种工作最好的地方就是activate()和deactivate()方法中。在EditPart接收到来自模型元素的事件的时候，它首先要判断这个事件对应的处理动作是简单的属性修改还是模型内部结构的变动。如果只是属性修改，只需要调用refreshVisual()方法，如果是模型内部结构的修改就需要调用refreshChildren()方法。

在这里还需要说明的是EditPart和模型元素之间这种基于事件的通信方式的依赖关系。例如，当一个连接改变了连接的EditPart的时候，新连接的EditPart和原来连接的EditPart都需要更新，要做的是确定如何更新这两个EditPart。在这种情况下，所有受影响的EditPart都需要fire对应类型的事件。

实现IPropertySource接口使得模型对象能够作为标准样式的属性页显示属性的来源。getEditableValue()返回实现了IPropertySource接口可以被作为标准属性页来源的对象，一般情况下返回模型对象本身。代码如下所示。

```
public Object getEditableValue() {
    return this;
}
```



getPropertyDescriptors()返回一个属性描述列表，列表中包含IPropertyDescriptor属性描述对象，其中包含了属性页中显示的属性名称。代码如下所示(代码见工程HelloGEF，hellogef.model.NodeModel.java)。

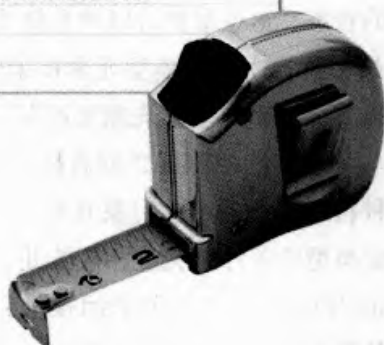

```
protected IPropertyDescriptor[] descriptors = new IPropertyDescriptor[] {
    new TextPropertyDescriptor(PROP_NAME, "模板名称"),
    new TextPropertyDescriptor(PROP_DESCRIPTION, "描述"),
    new TextPropertyDescriptor(PROP_AUTHOR, "作者"),
    new TextPropertyDescriptor(PROP_VERSION, "版本"),
    new TextPropertyDescriptor(PROP_LANGUAGE, "语言"),
    new TextPropertyDescriptor(PROP_MODIFY_TIME, "修改时间"),
    new TextPropertyDescriptor(PROP_CREATE_TIME, "创建时间")
};

public IPropertyDescriptor[] getPropertyDescriptors() {
    return descriptors;
}
```

Property	Value
版本	1.0
创建时间	2007-05-28 11:23:31.093
描述	
模板名称	测试_01
修改时间	2007-06-06 15:30:18.046
语言	简体中文
作者	

```
public Object getPropertyValue(Object id) {
    if (PROP_NAME.equals(id))
        return getName();
    if (PROP_DESCRIPTION.equals(id))
        return getDescription();
    if (PROP_AUTHOR.equals(id))
        return this.getAuthor();
    if (PROP_VERSION.equals(id))
        return this.getVersion();
    if (PROP_LANGUAGE.equals(id))
        return this.getLanguage();
    if (PROP_MODIFY_TIME.equals(id))
        return this.getModifyTime().toString();
    if (PROP_CREATE_TIME.equals(id))
        return this.getCreateTime().toString();
    return "";
}

public void setPropertyValue(Object id, Object value) {
    if (PROP_NAME.equals(id))
        setName((String) value);
    if (PROP_DESCRIPTION.equals(id))
        setDescription((String) value);
    if (PROP_AUTHOR.equals(id))
        this.setAuthor((String) value);
    if (PROP_VERSION.equals(id))
        this.setVersion((String) value);
    if (PROP_LANGUAGE.equals(id))
        this.setLanguage((String) value);
}
```



当名称属性被修改后

根据树形id判断属性值类型，
并将属性值设置到模型中

getPropertyValue()和setProperty()方法分别负责返回参数id对应模型属性的值和将属性页中修改的值设置到对应的模型属性。包含这些代码的模型就可以实现标准属性页与模型之间的交互操作了,其余的工作都会由GEF其他部分完成。

22.4.8 添加大纲视图——Outline

大纲视图是用来以树形结构显示GEF编辑器中图形元素之间层次关系的视图,主要用来显示图形中的层次嵌套结构辅助图形编辑。图22-41中把大纲视图图形元素关联编辑区域图形元素。



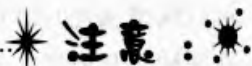
图22-41 大纲视图与图形元素的对应关系

创建大纲视图首先需要提供一个实现IContentOutlinePage接口的大纲视图页面类。下面OutlinePage类是扩展自ContentOutlinePage的大纲视图页面类的实现代码(代码见工程HelloGEF, hellogef.editors.HelloEditor.java)。

```
class OutlinePage extends ContentOutlinePage {
    private Control outline;
    public OutlinePage() {
        super(new TreeViewer());
    }
    public void init(IPageSite pageSite) {
        super.init(pageSite);
    }
    public void createControl(Composite parent) {
        outline = getViewer().createControl(parent);
    }
    public Control getControl() {
        return outline;
    }
    public void dispose() {
        getSelectionSynchronizer().removeViewer(getViewer());
        super.dispose();
    }
}
```

初始化大纲视图,并传入大纲视图所在的pageSite引用

创建大纲视图控件,并添加到父容器EditPartViewer中

**注意：**

这里的OutlinePage类不是从org.eclipse.ui.views.contentoutline.ContentOutlinePage派生的，而是从org.eclipse.gef.ui.parts.ContentOutlinePage派生的。

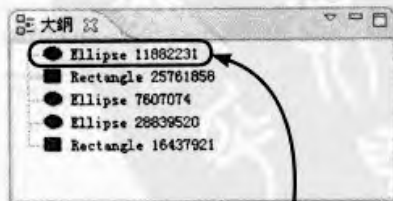


在前面OutlinePage的构造函数中，继承了GEF中的TreeViewer，这个TreeViewer 其实应该有相应的Editpart来反映模型的改变。GEF已经提供了org.eclipse.gef.editPart.AbstractTreeEditPart，需要从它派生创建一个DiagramTreeEditPart类，代码如下所示(代码见工程HelloGEF，hellogef.editors.DiagramTreeEditPart.java)。

```
public class DiagramTreeEditPart extends
AbstractTreeEditPart implements PropertyChangeListener{
    public DiagramTreeEditPart(Object model) {
        super(model);
    }
    public void propertyChange(PropertyChangeEvent evt) {
        refreshChildren();
    }
    public void activate() {
        // 启用EditPart，并将它注册到属性修改监听器中
        super.activate();
        ((Diagram) getModel()).addPropertyChangeListener(this);
    }
    public void deactivate() {
        // 禁用EditPart，并将它从属性修改监听器中注销
        super.deactivate();
        ((Diagram) getModel()).removePropertyChangeListener(this);
    }
    protected List getModelChildren() {
        // 返回EditPart中的所有子节点
        return ((Diagram) getModel()).getNodes();
    }
}
```

对应于节点模型NodeModel，还需要创建NodeTreeEditPart，代码如下所示(代码见工程HelloGEF，hellogef.editors.NodeTreeEditPart.java)。

```
public class NodeTreeEditPart extends
AbstractTreeEditPart implements PropertyChangeListener {
    public NodeTreeEditPart(Object model) {
        super(model);
    }
    public void propertyChange(PropertyChangeEvent evt) {
        refreshVisuals();
    }
    public void activate() {
        super.activate();
        ((NodeModel) getModel()).addPropertyChangeListener(this);
    }
    public void deactivate() {
        super.deactivate();
        ((NodeModel) getModel()).removePropertyChangeListener(this);
    }
    protected void refreshVisuals() {
        // 根据NodeModel模型NodeTreeEditPart对应的图形元素样式
    }
}
```



```

        setWidgetText(((NodeModel) getModel()).getName());
    }
}

```

与绘制图形类似，这里同样需要提供一个工厂类，把这些TreeEditPart和相应的模型联系在一起，代码如下所示(代码见工程HelloGEF，hellogef.editors.TreePartFactory.java)。

```

public class TreePartFactory implements EditPartFactory{
    public EditPart createEditPart(EditPart context, Object model) {
        if (model instanceof Diagram) {
            return new DiagramTreeEditPart(model);
        }
        else if (model instanceof NodeModel) {
            return new NodeTreeEditPart(model);
        }
        else {
            return null;
        }
    }
}

```

大纲视图中的DiagramTreeEditPart和NodeTreeEditPart分别对应于用户创建的模型元素

而后再回到OutlinePage中，添加设置并把图形绘制在大纲视图中，过程和绘制图形类似，代码如下所示(代码见工程HelloGEF，hellogef.editors.HelloEditor.java)。

```

class OutlinePage extends ContentOutlinePage {
    private Control outline;
    public OutlinePage() {
        super(new TreeViewer());
    }
    public void init(IPageSite pageSite) {
        super.init(pageSite);

        ActionRegistry registry = getActionRegistry();
        IActionBars bars = pageSite.getActionBars();
        String id = IWorkbenchActionConstants.UNDO;
        bars.setGlobalActionHandler(id, registry.getAction(id));
        id = IWorkbenchActionConstants.REDO;
        bars.setGlobalActionHandler(id, registry.getAction(id));
        id = IWorkbenchActionConstants.DELETE;
        bars.setGlobalActionHandler(id, registry.getAction(id));
        bars.updateActionBars();

    }
    public void createControl(Composite parent) {
        outline = getViewer().createControl(parent);

        getSelectionSynchronizer().addViewer(getViewer());
        getViewer().setEditDomain(getEditDomain());
        getViewer().setEditPartFactory(new TreePartFactory());
        getViewer().setContents(getDiagram());
    }
}

```

在初始化方法中向大纲视图页面所在的视图的ActionBar中添加Undo、Redo和delete按钮

初始化大纲视图，将视图添加到选择同步器中，并向视图中设置EditDomain、TreePartFactory工厂类和模型


```

    }
    public Control getControl() {
        return outline;
    }
    public void dispose() {
        getSelectionSynchronizer().removeViewer(getViewer());
        super.dispose();
    }
}

```

将视图从选择同步器中移除

完成以上工作之后，只需要在HelloEditor中的getAdapter(Class type)添加下面框中的代码之后，大纲视图就可以工作了。

```

public class HelloEditor extends GraphicalEditorWithFlyoutPalette {
    public Object getAdapter(Class type) {
        if (type == IContentOutlinePage.class)
            return new OutlinePage();

        return super.getAdapter(type);
    }
}

```

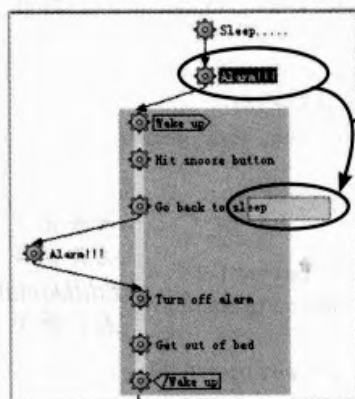
※ 注意: ※

大纲视图是辅助显示各种图形元素之间层次结构的视图，如果图形编辑器中不允许用户创建诸如多层图形嵌套等复杂图形的情况就没有必要创建大纲视图。



22.4.9 实现拖放功能 (Drag and drop)

图形的拖放是可视化图形编辑桌面应用的主要操作方式，下面来介绍如何在基于GEF框架的应用中加入拖放功能。在SWT中拖放操作的核心是Transfers，它表示了SWT控件之间通过拖放操作传递的对象。GEF中的拖放类似于SWT。图22-42为图形元素拖动效果，在拖动过程中会产生一个与原图形元素相同大小，并随鼠标移动的阴影以表示拖动后图形元素所占据的位置。



鼠标在GEF应用中的编辑界面上拖动一个图形元素



图22-42 拖动一个图形元素

在GEF中提供了一些拖放操作的默认实现类，简化了实现过程。如读者不需要关心SWT DragSource对象和其他底层实现类。具体实现过程是首先向EditPartViewer中添加拖放Transfer源监听器TransferDragSourceListener和拖放Transfer目的监听器TransferDropTargetListener。TransferDragSourceListener用来定义EditPartViewer作为拖放源响应拖放操作，而TransferDropTargetListener用来定义EditPartViewer作为拖放目的响应拖放操作。在向GEF应用加入拖放功能的时候，拖放监听器的实现必须是继承自包含默认实现的GEF中提供的抽象类。

22.4.10 放大缩小 (Zooming)

GEF中的放大缩小功能也是在2.0版本中提供的。Scaling功能添加到了ScalableLayeredPane和ScalableFreeformLayeredPane之中。这些类通过维护当前图形尺寸和放大缩小参数计算显示区域的位置大小，并使用Graphics的子类ScaledGraphics作为Graphics context绘制包含的child Figure。

ScaledGraphics类参照当前scale factor完成Graphics的操作，在绘制图形之前它会自动完成坐标和大小的转换。它也可以放大缩小字体，图标和线条。

ScalableLayeredPane和ScalableFreeformLayeredPane中包含的ScalableFigure类的Figure图形元素缩放操作由ZoomManager管理，ZoomManager类中提供了一些控制缩放操作的方法，如下所示。

- ★支持设置放大缩小方式，目前提供的方式有跳跃式和动态放大缩小。
- ★设置放大缩小等级。
- ★设置视图位置。
- ★支持zoom listeners，所有支持放大缩小功能的控件都可以注册为zoom listeners。

在ScalableFreeformRootEditPart 和ScalableRootEditPart中包含ZoomManager，通过getZoomManager()方法可以获取。通过editor的GraphicalViewer获取ZoomManager的代码如下所示。

```
ZoomManager zoomManager =
((ScalableFreeformRootEditPart)getGraphicalViewer().getRootEditPart()).getZoomManager()
```

如果需要修改ZoomManager的配置，可以采用上面的方法。当向editor界面上添加放大缩小控件的时候，GEF中提供有ZoomComboContributionItem类用来创建zoom actions和ContributionItem。

ZoomComboContributionItem创建了以上样式的放大缩小比例设置控件，并用IAdapter接口在IWorkbenchPart中查找ZoomManager。代码如下所示。

```
public Object getAdapter(Class type) {
    if (type == ZoomManager.class)
        return((ScalableFreeformRootEditPart)getGraphicalViewer().
            getRootEditPart()).getZoomManager();
    return null;
}
```

GEF中提供的ZoomInAction和ZoomOutAction简化了向菜单栏加入放大缩小菜单项和向工具栏加入放大缩小按钮的过程。这两个actions都继承自ZoomAction类，ZoomAction类保存当前的

ZoomManager并将自己注册为ZoomListener。这些actions在被激活时调用ZoomManager.zoomin()或ZoomManager.zoomOut()放大或缩小图形。通过实现ZoomListener接口的zoomChanged()方法可以监控到是否达到放大缩小最大值，并设置放大或缩小按钮菜单是否可用。

在向GEF应用中添加放大缩小功能之前，必须确保root EditPart是ScalableFreeformRootEditPart或者ScalableRootEditPart。这时只需向editor中添加放大缩小界面控件就可以了。向工具条中添加放大缩小控件需要先把控件添加到实现了ActionBarContributor和IToolBarManager接口的工具条管理器。添加放大缩小功能的具体代码如图22-43所示。

```
public void contributeToToolBar(IToolBarManager toolBarManager)
{
    super.contributeToToolBar(toolBarManager);
    //other items added here.....
    toolBarManager.add(new Separator());

    toolBarManager.add (new ZoomComboContributionItem(getPage()));
}
```



图22-43 带缩放控件的工具栏

向继承自EditorPart的editor中注册zoom actions的方法如下所示。

```
IAction zoomIn = new ZoomInAction(zoomManager);
IAction zoomOut = new ZoomOutAction(zoomManager);
getActionRegistry().registerAction(zoomIn);
getActionRegistry().registerAction(zoomOut);
//also bind the actions to keyboard shortcuts
getSite().getKeyBindingService().registerAction(zoomIn);
getSite().getKeyBindingService().registerAction(zoomOut);
```

向MenuManager中添加zoom actions对应的菜单项的方法如图22-44所示。

```
public void contributeToMenu(IMenuManager menuManager) {
    super.contributeToMenu(menuManager);
    //add a "View" menu after "Edit"
    MenuManager viewMenu = new MenuManager("View");
    viewMenu.add(getAction(GEFAActionConstants.ZOOM_IN));
    viewMenu.add(getAction(GEFAActionConstants.ZOOM_OUT));
}
```

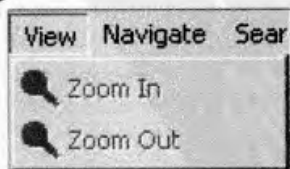


图22-44 缩放菜单栏

22.4.11 添加直接编辑功能

直接编辑功能是GEF框架中提供的通过在指定位置显示编辑框编辑图形属性的功能。触发直接编辑功能的方式有如下几种。

- ★在图形元素上双击鼠标。
- ★在已经选中的图形元素上单击鼠标。
- ★选中需要编辑的图形元素，而后按F2键。

图22-45中显示了直接编辑功能的编辑样式。直接编辑功能也可以用在复杂的编辑场景下，如使用其他属性编辑器、自定义弹出框编辑属性等。直接编辑功能的触发动作是可以自定义的，不但可以指定自定义弹出编辑器，而且可以设置响应规则，如依据单击图形元素的具体位置或图形元素对应模型属性值弹出不同编辑器等。

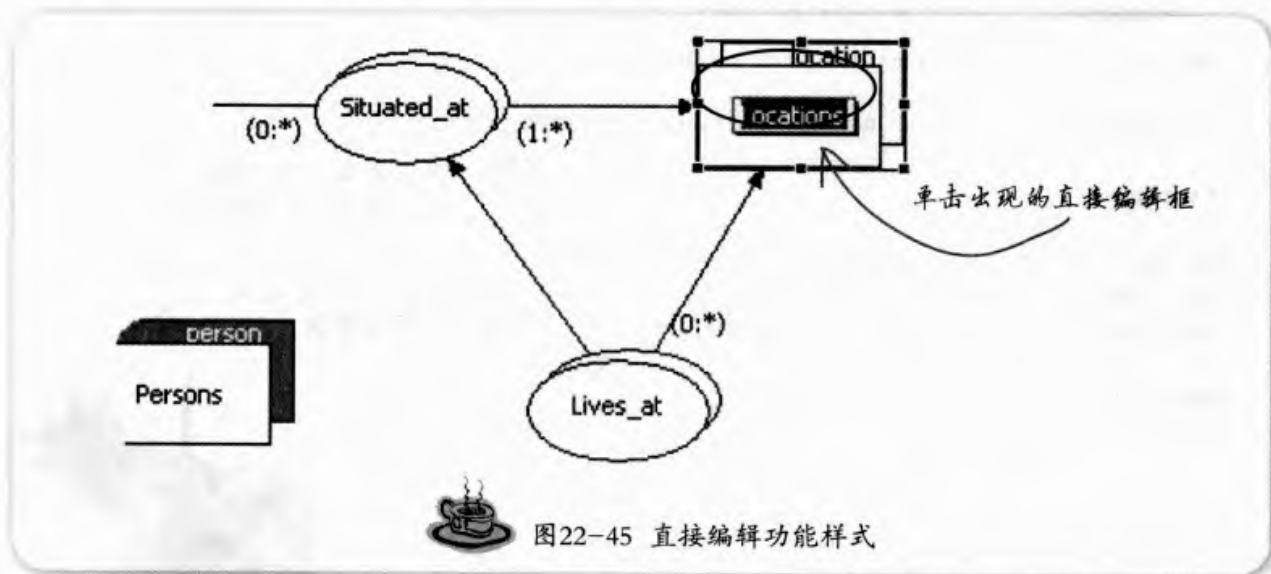


图22-45 直接编辑功能样式

Eclipse中提供的编辑器默认实现是org.eclipse.jface.viewers.CellEditor。所有自定义的编辑器都要继承它。Eclipse中还包含了它的一些子类实现，如布尔值编辑器、下拉列表框编辑器、文本编辑器和对话框编辑器等。通过继承DialogCellEditor，可以创建弹出框式编辑器，灵活地编辑图形元素属性。下面是实现直接编辑功能的具体过程。

首先修改EditPart中的createEditPolicy()方法，加入EditPolicy.DIRECT_EDIT_ROLE键对应的策略，而后向performRequest()方法中添加显示继承自DirectEditManager的子类编辑管理器的代码。如果没有编辑管理器则创建之。在创建过程中，创建函数的第二个参数指定了要创建什么类型的CellEditor，第三个参数则需要传入编辑器的定位器，它是继承自org.eclipse.gef.tools.CellEditorLocator的子类。定位器将计算编辑器在选中图形元素中显示的具体位置。示例代码中使用的对应于直接编辑请求的键为RequestConstants.REQ_DIRECT_EDIT。请求值将决定使用什么方式触发直接编辑功能。RequestConstants.REQ_DIRECT_EDIT表示用鼠标单击选中图形元素的方式触发请求，RequestConstants.REQ_OPEN则表示用双击方式触发请求，也可以在自定义的GEF应用中同时采用两种触发方式。代码如下所示(代码见工程HelloGEF，hellogef.control.NodePart.java)。


```
public class NodePart extends AbstractGraphicalEditPart implements
PropertyChangeListener, NodeEditPart {
protected DirectEditManager manager;
protected void createEditPolicy(){
    super.createEditPolicy();
    installEditPolicy(...);
}
```

向EditPart中添加直接编辑策略 - LabelDirectEditPolicy

```
installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new
LabelDirectEditPolicy());
```

```
}
public class NodePart extends AbstractGraphicalEditPart implements PropertyChangeListener,
NodeEditPart {
    protected DirectEditManager manager;
    public void performRequest(Request request){
```

```
if (request.getType() == RequestConstants.REQ_DIRECT_EDIT) {
```

```
if(manager == null)
```

```
manager = new LogicLabelEditManager(this,
TextCellEditor.class, new
LabelCellEditorLocator((Label) getFigure()));
```

```
manager.show();
```

```
}
```

```
}
```

用鼠标单击选中图形元素的方式触发请求

创建以文本框形式编辑的直接编辑框



在EditPart中创建的对应用于直接编辑功能的策略必须是org.eclipse.gef.editPolicy.DirectEditPolicy类的子类。策略中至少需要提供以下两个方法的实现。

★protected abstract Command getDirectEditCommand(DirectEditRequest request): 这个方法创建直接编辑请求对应的Command。在Command中会根据直接编辑后的值修改模型属性，也可以Undo/Redo这个过程。

★protected abstract void showCurrentEditValue(DirectEditRequest request): 根据请求中直接编辑后的值更新对应于EditPart的Figure。

在图形上直接编辑图元时，通常情况下都需要先选中要编辑的图元，处于选中状态的图元会显示边框和控制点，这有可能影响直接编辑框的显示。要解决这个问题需要做以下几个工作。

首先在DirectEditManager的子类的show()方法实现中保存当前选中状态，方法为List savedSelection = source.getSelectedEditPart(); 然后暂时取消图形上的选中source.deselectAll()。重写bringDown()方法确保保存的状态在直接编辑完成后恢复。

22.4.12 其他相关内容

下面介绍了创建GEF应用的过程中有可能用到的一些其他相关内容，这些内容包括了GEF应用中的资源管理，图形界面中的可视化反馈和打印功能。

1. 资源管理

在创建GEF应用的时候，注意管理应用中对图形系统资源的使用情况是很重要的。读者必须维护包括图标、图片、颜色、字体在内的一系列图形显示相关对象。下面列举了帮助管理应用中这些资源的一些常用技术。

为在GEF应用整个生命周期中都会被频繁使用的资源创建静态变量。例如颜色管理，org.eclipse.draw2d.ColorConstants中就提供了很多有用的静态常量。

需要在EditPart生命周期中动态创建的图形资源可以分别在EditPart类active()和deactivate()方法中创建释放这些图形资源。这些图形资源一般包括节点、连线等与EditPart对应的图形元素。

如果在应用中存在需要多个EditPart共用的图形资源，可以考虑采用缓存来管理这些图形对象，这样不同EditPart之间就可以使用同一个对象实例，节约内存空间。

2. 反馈

可视化反馈是图形编辑界面中图形元素展现、更新和模型属性定义以及模型更新之间同步的一个重要功能。GEF框架中允许开发者创建多种反馈机制提供给最终用户使用。如下所示。

★通过改变光标图标反馈用户目标图形是否支持当前选中的工具操作，或者是否支持拖放操作。

★指示当前图形选中和获取焦点状态。一般情况下，被选中的元素都应该改变显示样式以区别于其他同类的元素。通常会改变选中图形元素的颜色，形状或者用selection Figure图形元素包围选中图形。

★显示锚点，表示图形元素可以被移动或改变形状。

在GEF中，大部分反馈效果是有包含在EditPart中的各个EditPolicy控制的。下面就来讲解EditPolicy是如何控制这些图形反馈效果以及如何创建自定义反馈效果。

EditPolicy中控制反馈的主要是其子类GraphicalEditPolicy。这个类提供了访问EditPart中Figure的功能。其中也声明了用于在root Figure 的root feedback层(LayerConstants.FEEDBACK_LAYER)绘制反馈Figure的addFeedback()和removeFeedback()方法。

在EditPolicy中当接收到创建Command的请求时如果返回null就可以使鼠标光标显示禁用操作光标。在Command中的canExecute()方法中返回false也可以产生相同的反馈效果。

直接编辑策略(DirectEditPolicy)指的是当双击包含该策略的图形元素时，通过在图形元素上显示一个编辑框使得用户可以直接在图形界面修改选中图形元素所对应模型属性的策略。策略中允许自定义的部分有如下几个。

★可以自定义单击图形元素不同部分编辑不同的模型属性。

★自定义双击响应方式。

★允许创建自定义的弹出编辑器。

3. 打印

GEF的2.0版本之后, 提供了对打印编辑器editor中图形功能的支持。读者只需要向editor的action registry中添加PrintAction就可以实现GEF应用的打印功能。下面列出了添加PrintAction过程的代码。

```
ActionRegistry registry = getActionRegistry();
IAction action;
action = new PrintAction (this);
registry.registerAction(action);
```

这里的this 是继承自IEditorPart接口的editor对象

在GEF应用的action bar的contributor类中, 调用方法addGlobalActionKey(IWorkbenchActionConstants.PRINT)可以添加菜单栏和键盘调用PrintAction的功能。Draw2d中的PrintOperation类和其子类PrintFigureOperation, PrintGraphicalViewerOperation也提供了打印功能。

打印过程中, PrintGraphicalViewerOperation类首先找到editor viewer中的printable层。当前editor中选中的项被保存并disable, 在打印完成后这些选中项会被恢复到原始状态。注意, Editor viewer中所有的层次中, 只有printable类型的层次层才能被打印。Parent Figure 的背景会被设置为白色。打印所包含的Figure大小会依据打印分辨率和显示分辨率的比例来调整。

至此就完成了HelloGEF工程GEF应用的创建, 编辑器在Eclipse平台中的显示样式如图22-46所示。

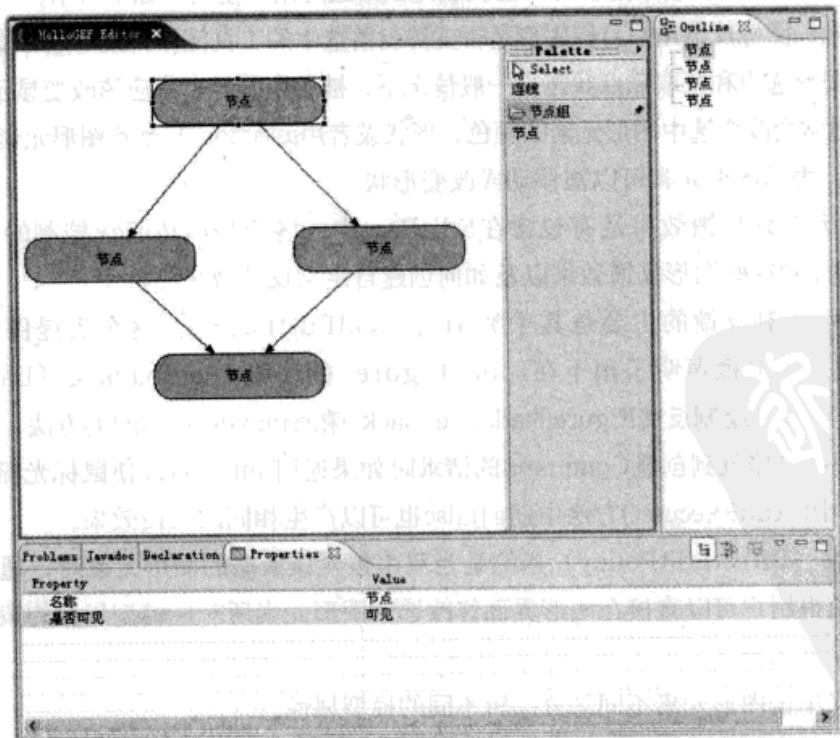
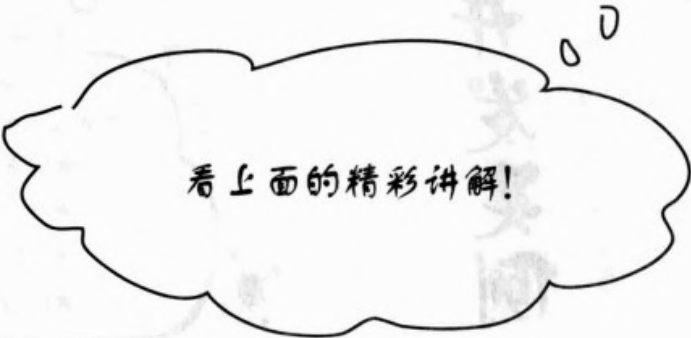


图22-46 HelloGEF编辑器样式

22.5 本章总结

GEF架构为创建基于Eclipse平台的图形编辑器提供了便利，大大简化了工具的创建过程。本章详细介绍了GEF架构的组成和运行原理，并通过实例讲解了创建GEF应用的具体步骤和实现方法，在本书实例章节将对实现GEF编辑器应用的完整过程做详细讲解。



看上面的精彩讲解!



第23章 插件开发实例

通过前面各章的学习，读者已经掌握了插件开发所需的理论知识。但仅有理论知识是不够的，在实际的开发过程中，如何根据实际需求，将这些知识灵活运用来解决问题和学习知识本身同样重要。本章将从需求分析开始，一步一步地设计并实现一个实用的插件产品，并尝试在已有的RCP程序中重用它的功能。通过本章的学习，读者可以了解如何设计结构灵活、易于重用的插件产品。阅读本章时，需要结合光盘上的示例代码。

本章内容包括：

- ★需求分析与设计。
- ★插件开发。
- ★在RCP程序中重用插件。
- ★FAQ。

拉开崭新的学习帷幕

进入第23章



23.1 需求分析与设计

在日常工作中，经常需要记录一些待办事项，如会议日程或工作安排等，虽然已经有很多现成的软件，如Outlook，可以用来完成这一工作，但是如果能够将这项功能集中到朝夕相处的Eclipse中，对开发者而言会更为方便。本着“自己动手，丰衣足食”的原则，本节将演示如何开发一个能够满足这些需求的插件——To-Dos。

23.1.1 需求整理

开发工作的第一步，从收集和整理需求开始。对该插件的需求，总结起来有如下几条。

1. 能够显示日历。事项的安排都是基于日期的，自然需要有日历的支持。
2. 可以在日历上为某一天添加事项列表。事项列表包括待办事项、日程安排等内容。有事项的日期在日历上会以粗体字显示，单击这个日期就可以看到当天所有的事件备忘，双击列表中某一项可以打开并编辑它的内容。具体如下所示。

★用户设定事件的日期时，可以选择具体的日期，也可以设定一个选择条件。如“每周二”、“每月25日”等。用户设定选择条件后，需要指定一个执行周期。可以指定周期的开始时间和结束时间，如从2007年8月1日开始，到2007年10月1日结束；也可以不指定周期的结束时间，使事件一直重复发生，这样创建的备忘事项被称为同一个系列的事项。

★修改一个系列中的某个事项时，可以选择同时修改整个系列的内容，也可以选择只修改选中的事项内容。如果对某一个事项选择“只修改选中的事项内容”并做出修改后，下一次选择“修改整个系列”时，这些已经单独修改过的事项不会受到影响。

3. 删除待办事项，删除一个系列的事项时，该系列中所有单独修改过的事项也会一起被删除。

23.1.2 数据模型抽象

了解了需求以后，需要进行建模，从需求中抽象出业务对象模型。在这个业务需求中，最核心的模型显然是待办事项，第一步的建模就从这里开始。待办事项类图如图23-1所示。



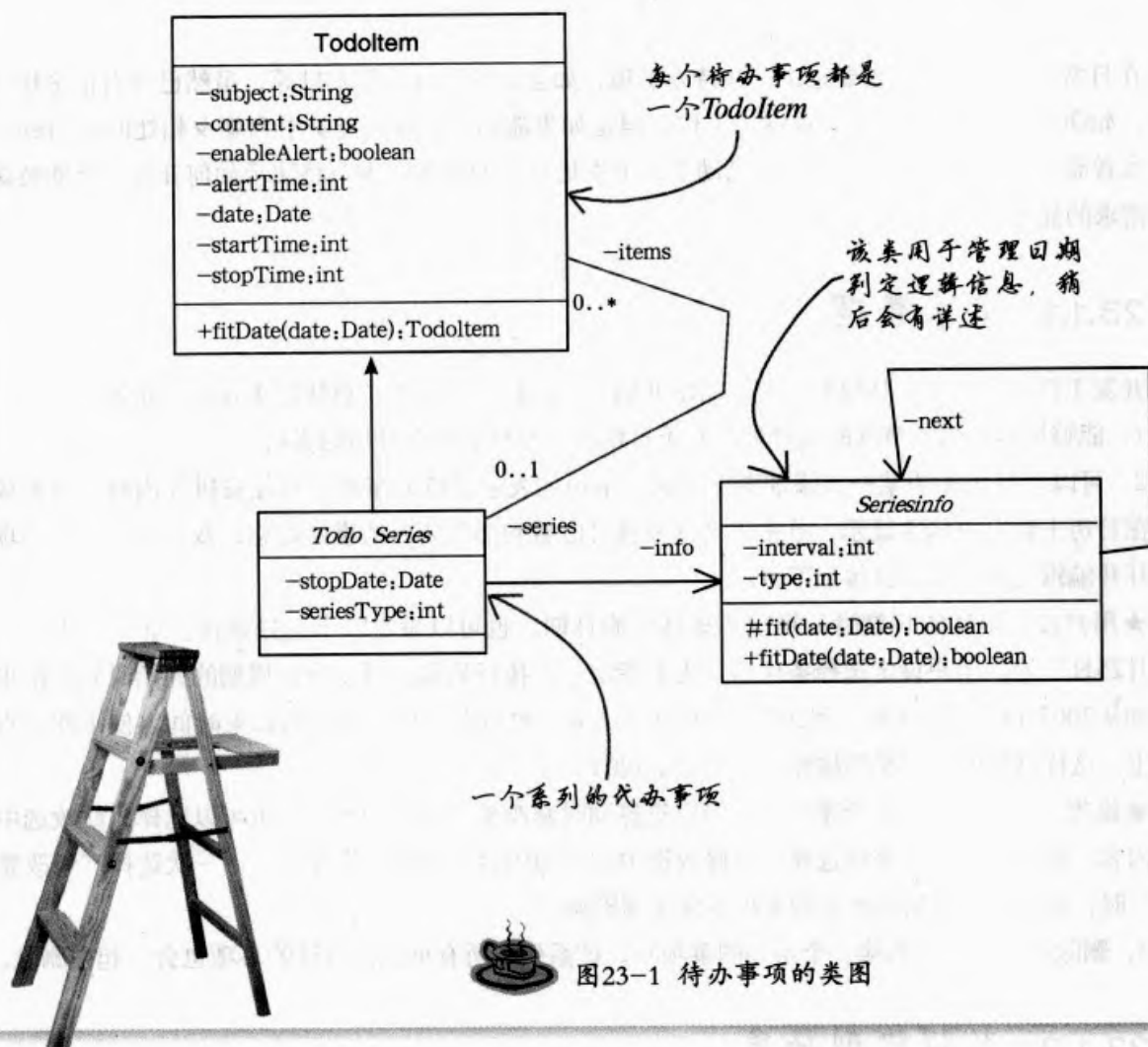


图23-1 待办事项类图

TodoItem类表示能够显示在待办事项列表中的一条项目，通过方法fitDate可以判定该TodoItem是否应该在指定日期对应的事项列表中显示。它包含了事项的标题、内容、开始时间和结束时间等基本信息。

TodoItem的子类TodoSeries代表了系列类型的事项，它的属性seriesType表示了系列的时间类型（有结束时间的类型和无结束时间的类型），而stopDate属性只在系列的类型为“有结束时间型”时有效，它代表了系列的结束时间。

TodoSeries中维护了一个TodoItem的队列，它用来记录系列中曾经被单独修改过的TodoItem信息。因此，用户在列表中看到的事项可能有以下三种类型。

- ★独立的一个TodoItem
- ★TodoSeries
- ★TodoSeries下面的一个TodoItem

图23-2显示了当用户试图对这三种类型的事项进行操作时，系统的处理的流程图。

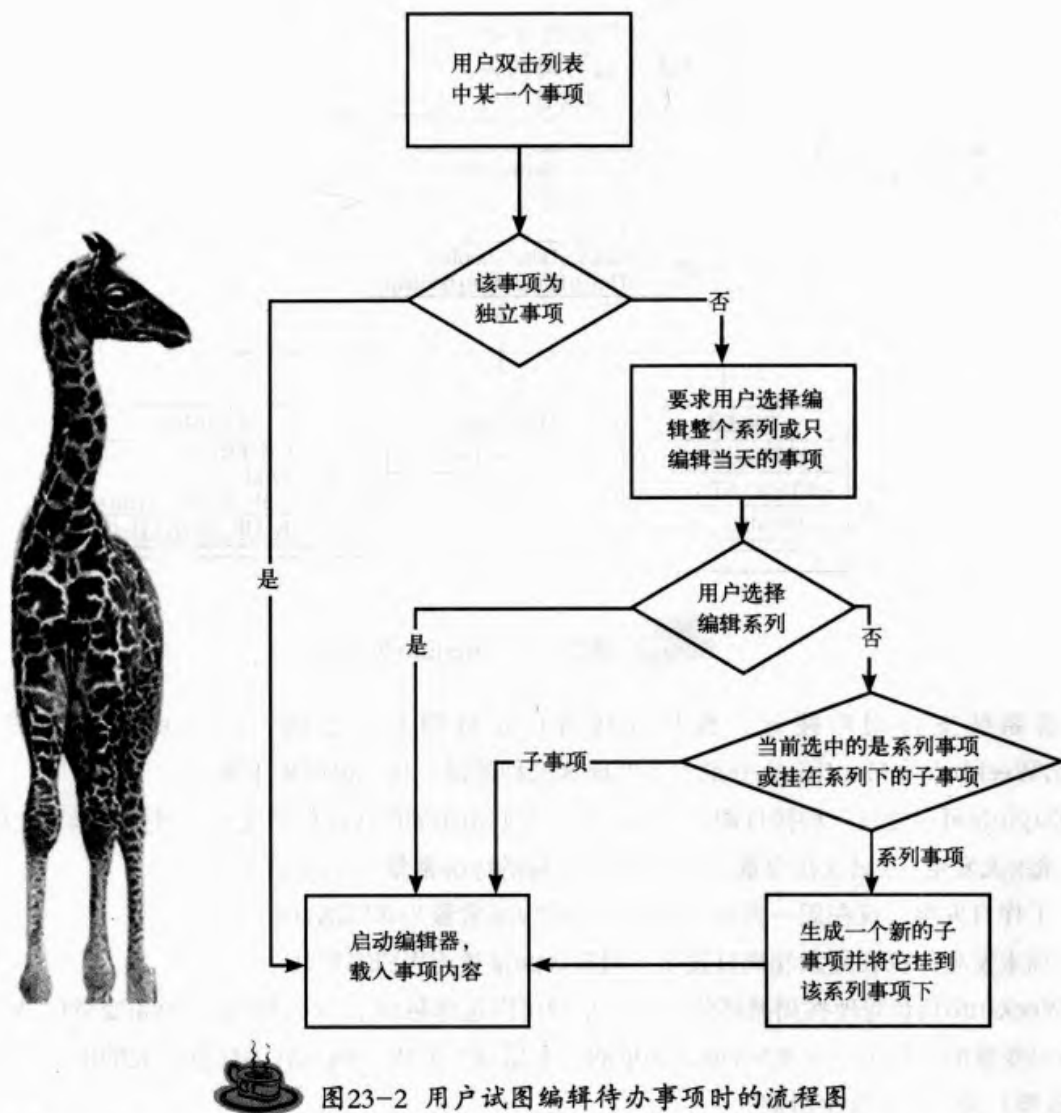


图23-2 用户试图编辑待办事项时的流程图

用户单击一个独立的TodoItem时，可以直接编辑这个TodoItem的内容；而当用户单击了一个TodoSeries，程序会要求用户选择希望只编辑当天的事件还是编辑整个系列，如果用户选择“编辑系列”，则直接修改系列的内容；如果用户选择“只编辑当天的事件”，则创建一个新的TodoItem并将TodoSeries的标题和文字内容复制到该Item，并打开编辑器编辑这个新事项，新事项的内容不会影响到原有的系列；如果用户选择的是关联到TodoSeries的TodoItem并试图编辑时，同样会要求在编辑系列和编辑当天事件之间做出选择，这种情况下如果选择“编辑系列”，则允许用户修改系列的内容，编辑结果不会影响当前选中的事项，而选择“编辑当天事件”则直接编辑选中的事项内容。

另外在修改时，用户只能改动事项的标题、内容和发生时间等信息，而不能改变事项类型，如把事件从系列类型改到普通类型等。

普通事项有唯一的发生日期，而系列事项则是通过日期判定逻辑来判别该系列的事项是否会在某一天发生，如“每周二”或“每月三号”等信息都属于日期判定逻辑，这些信息使用抽象类SeriesInfo类型存储与处理。通过SeriesInfo的fit(Date date)方法可以判定一个日期是否与该日期判定逻辑相吻合，SeriesInfo相关的类图如图23-3所示。

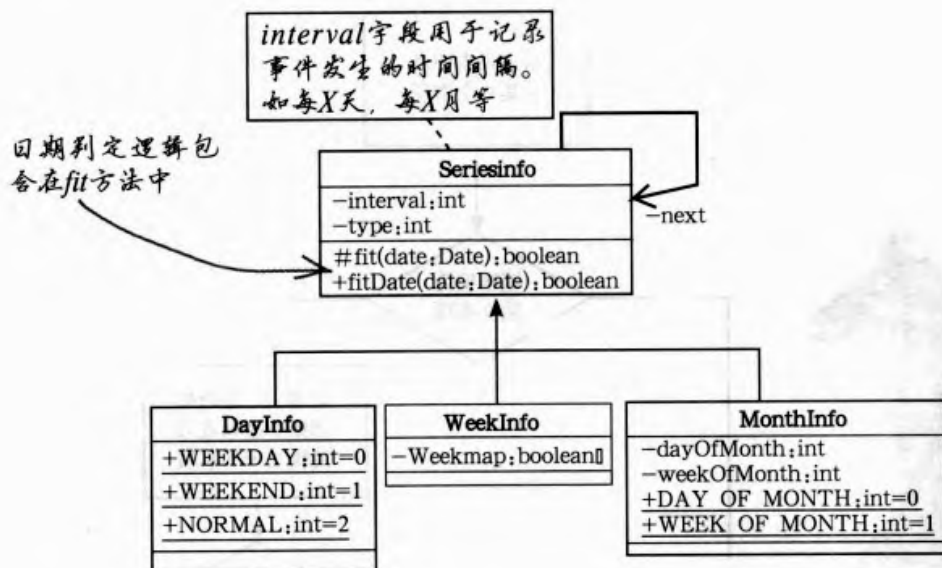


图23-3 SeriesInfo的类图

目前系统支持用户按日、按周或按月设定日期判定逻辑，SeriesInfo的三个子类DayInfo、WeekInfo和MonthInfo分别负责处理这三种逻辑。具体说明如下所示。

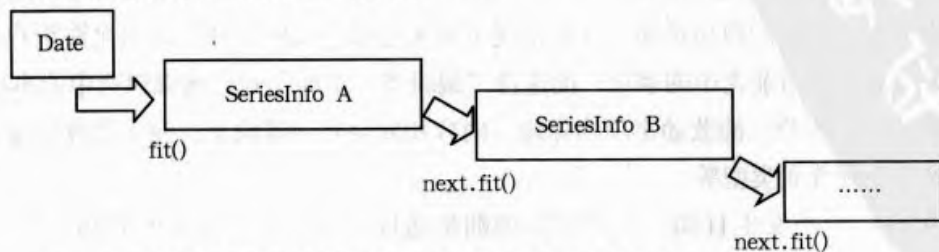
★DayInfo可以处理三种按日期循环的逻辑，它们由不同的type取值区分，可分为以下三种情况。

1. 每*x*天发生。*x*记录在变量*interval*中，对应的type常量为NORMAL。
2. 工作日发生。仅在周一到周五发生，对应type常量为WEEKDAY。
3. 周末发生。仅在周六和周日发生，对应type常量为WEEKEND。

★WeekInfo负责管理按周循环的周期。用户可以选择每*x*周（*x*同样由*interval*变量记录，这也是将*interval*变量的声明放在父类SeriesInfo里的一个原因）的周一到周日中任意几天的组合（如选择周一和周五等）为事件的发生时间。

★MonthInfo允许用户指定事件在每*x*（*x*由*interval*指定）个月内的第*y*天发生（*y*由变量*dayOfMonth*指定）。

为了处理更为复杂的日期判定模式，也为了使模型更简练，SeriesInfo中采用了责任链（Responsibility Chain）的设计模式。开发者可以将多个不同的SeriesInfo对象使用next关联组合起来，以得到更复杂的判断逻辑，如图23-4所示。



Date信息会依次接受责任链上所有SeriesInfo对象的判定



图23-4 SeriesInfo中的责任链

如将一个代表“每月第三周”的SeriesInfo对象(具体地说,是MonthInfo对象)和代表“工作日”(DayInfo)的对象串联起来,就可以得到“每月第三周的工作日”这样的判断逻辑,在SeriesInfo的fitDate(Date date)方法中,会逐个调用一条链上每个SeriesInfo对象的fit方法,如果其中有任何一个的fit方法返回false,则fitDate方法返回false;反之则返回true。

TodoSeries持有一个SeriesInfo对象的引用,在它判断某个日期是否包含在指定的系列中时,会调用fitDate方法来判定该日期是否与SeriesInfo的要求相符合。

23.1.3 体系架构和插件结构

得到了作为系统核心的业务模型抽象后,下一步工作是选择项目的体系架构,进而确定插件的结构。

因为该插件产品是面向单用户的本地程序,暂时不考虑服务器的支持。那么首先需要为程序选择一种持久化解决方案(简单来说,就是如何存储和读取数据)。考虑到事项记录结构比较分散,数量比较大的特点,决定使用本地的MySQL数据库提供持久化服务,并使用Oracle™TopLink提供对象-关系型数据映射服务,结构如图23-5所示。

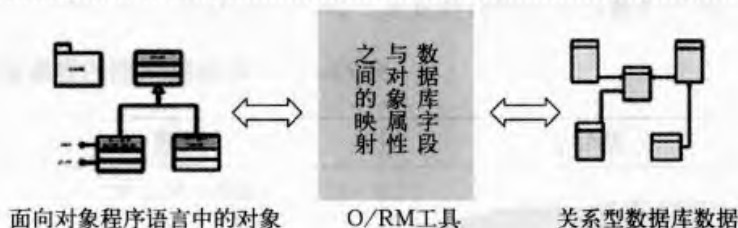


图23-5 系统架构图



※ 注意: ※

对象-关系型数据映射(Object-Relational Mapping, O/RM)是一种将关系型数据库中的数据与面向对象编程语言中的对象相互转换的技术,用户的程序可以直接操作对象,而O/RM的实现框架则将这些操作转化成SQL语句传到关系型数据库来执行,使用这种技术,可以基于已经广泛应用的关系型数据库,为面向对象的程序语言创建一个虚拟的面向对象数据库。如下图所示。



目前在Java语言开发中可以使用的、较成熟的O/RM工具主要有Oracle™TopLink和Hibernate。对于非商业性的开发活动,它们都是完全免费的。

程序的UI界面和业务逻辑都要基于插件体系开发,下面将对这一部分进行设计,这包含应该创建几个插件以及划分它们各自的职责等内容,最简单的结构是将所有用到的内容,包括数据模型和UI部分统统放在一个插件里面。初学者偏爱这种做法,因为它配置起来比较简单;而且相较于多个插件的结构而言,不需要耗费太多精力在插件功能的划分和依赖关系的安排上。然而作者个人比较推荐的方法是将整个程序按照数据模型、UI的分类分为两部分。这样当一部分发生改动,也不必影响其他部分(如Eclipse的下一个版本推出了更炫的界面组件,因此希望重新开发插件的UI);另外在有多人共同开发时,可以方便地实现并行开发;而且这也比较符合模块化程序设计“高内聚,低耦

合”的要求。

除此之外,考虑到在程序中使用到TopLink,MySQL JDBC驱动等第三方包的需求(关于在插件或RCP程序中使用第三方软件包的内容,可以参见本章FAQ一节中的相关内容),最后决定的插件设计方案如图23-6所示。

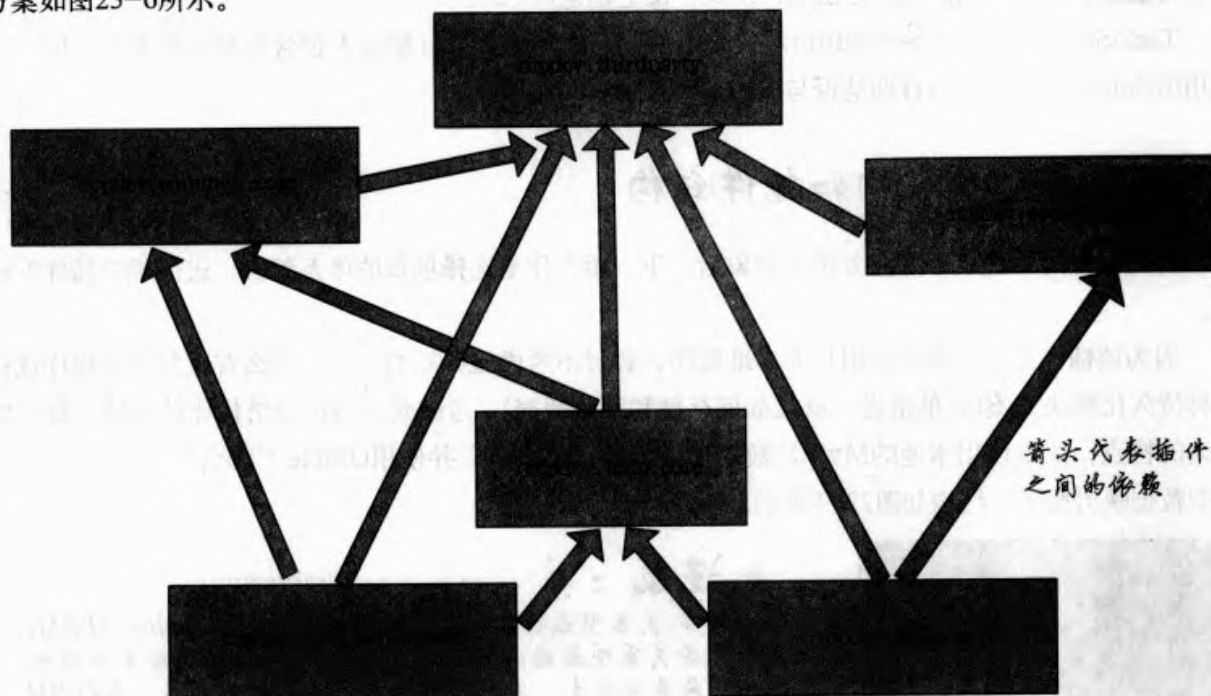


图23-6 插件的结构依赖图

图23-6看起来略微有些复杂,表23-1对其中所包含插件的作用作了详细描述。

表23-1 对插件结构图的详细说明

插件名	说明	其他依赖
rcpdev.thirdparty	包含了插件中可能用到的所有第三方包	org.eclipse.core.runtime
rcpdev.common.core	具有通用性的、与界面无关的组件	org.eclipse.core.runtime
rcpdev.common.ui	具有通用性的、与界面相关的组件	org.eclipse.core.runtime org.eclipse.ui
rcpdev.todo.core	To-Dos插件的核心部分,包含了数据模型和持久化接口的声明	org.eclipse.core.runtime
rcpdev.todo.ui	To-Dos插件的UI部分,包含了在插件中用到的视图、编辑器、向导页等	org.eclipse.core.runtime org.eclipse.ui
rcpdev.todo.persistence	为To-Dos插件提供持久化解决方案(使用MySQL数据库存储),它通过扩展点向todo.core插件提供持久化接口的一个实现	org.eclipse.core.runtime org.eclipse.ui

插件中用到第三方JAR包由单独的插件rcpdev.thirdparty维护,另外,在开发过程中,常常编写一些功能类代码和通用的组件,将它们从程序里剥离出来单独维护和管理是一个好习惯,日后开发其他程序时,也可以方便地重用这些代码,本插件示例程序中,将与界面无关的组件和代码放到了rcpdev.common.core插件中,而将为界面开发服务的部分以及部分JavaBean组件归到了rcpdev.common.ui插件中。

除这些公共插件外,示例程序To-Dos由三个插件组成,其中,除包含核心数据模型的rcpdev.todo.core插件和包含UI元素的rcpdev.todo.ui插件外,将持久化部分rcpdev.todo.persistence也独立出来作为单独的插件提供,这种设计给予程序更大的灵活性,如果用户不希望使用数据库。而希望使用文件或其他方式作为持久化解决方案,也只需要开发一个新版本的persistence插件就可以方便的实现这一要求,如图23-7所示。

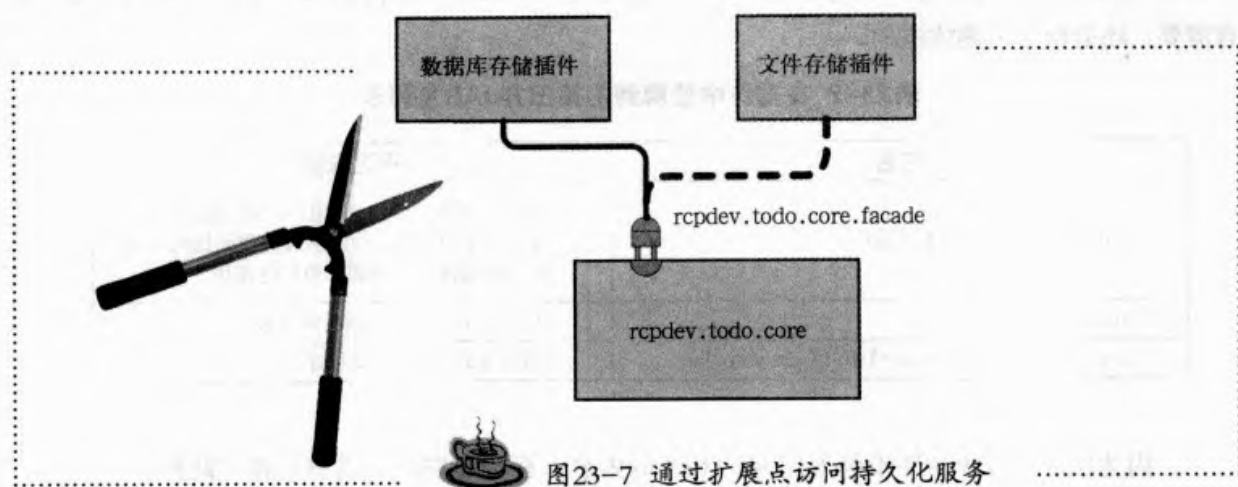


图23-7 通过扩展点访问持久化服务

最后要选择合适的UI载体来承担各项功能,日历和备忘列表需要能够供用户随时查询,因此使用视图来实现;创建对象时,可以使用向导,而编辑事项的工作,自然是交给编辑器来完成了。现在,可以画出如图23-8所示的结构示意图。

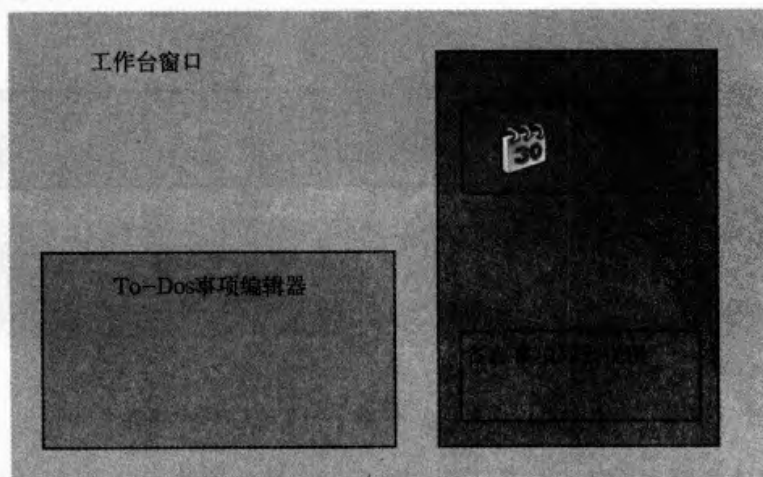


图23-8 插件的UI设计

在完成了设计部分后，对整个项目的轮廓已经有了大致的把握。下面，就可以着手开发了。

23.2 插件开发

本节将对组成示例程序的各个插件的内容做概要的叙述，这可以作为读者阅读示例程序代码的索引。

23.2.1 插件rcpdev.thirdparty

插件rcpdev.thirdparty中目前共包含了三个第三方的JAR包，如表23-2所示，如果在开发工作中有需要，还会向其中继续添加。

表23-2 在插件中使用到的第三方JAR包列表

包名	用途说明
commons-jxpath-1.2.jar	JXPath的运行时。使用JXPath提供的API，可以用类似XPath的语法来方便地访问Java对象的属性。在本插件中广泛使用
toplink.jar	Oracle TopLink的运行时环境
mysql-connector-java-3.0.15-ga-bin.jar	MySQL的JDBC驱动程序

因为thirdparty插件不包含任何UI相关的内容，在创建它时，没有选择“此插件将对UI进行添加”，如图23-9所示，这时插件的依赖列表中不包含org.eclipse.ui插件，同时所生成的激活器Activator也将继承自类型Plugin，而不是UI相关插件的激活器所继承的AbstractUIPlugin。后者继承自前者，并额外提供了部分与图形界面相关的功能，如提供一个用于存储首选项的IPreferenceStore对象以及一个用于缓存Image对象的ImageRegistry等。

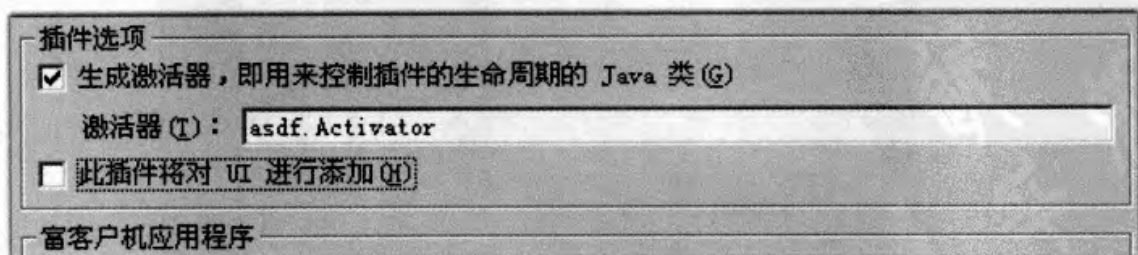


图23-9 创建一个UI无关的插件

将三个JAR文件复制到插件根目录下面，然后打开插件清单，在“运行时”选项卡中的“类路径”部分将这三个文件加入到类路径里。随后在“已导出的包”部分将所有列出的包加入导出列表。现在，任何依赖于rcpdev.thirdparty的其他插件都可以直接使用这些第三方包了，如图23-10所示。

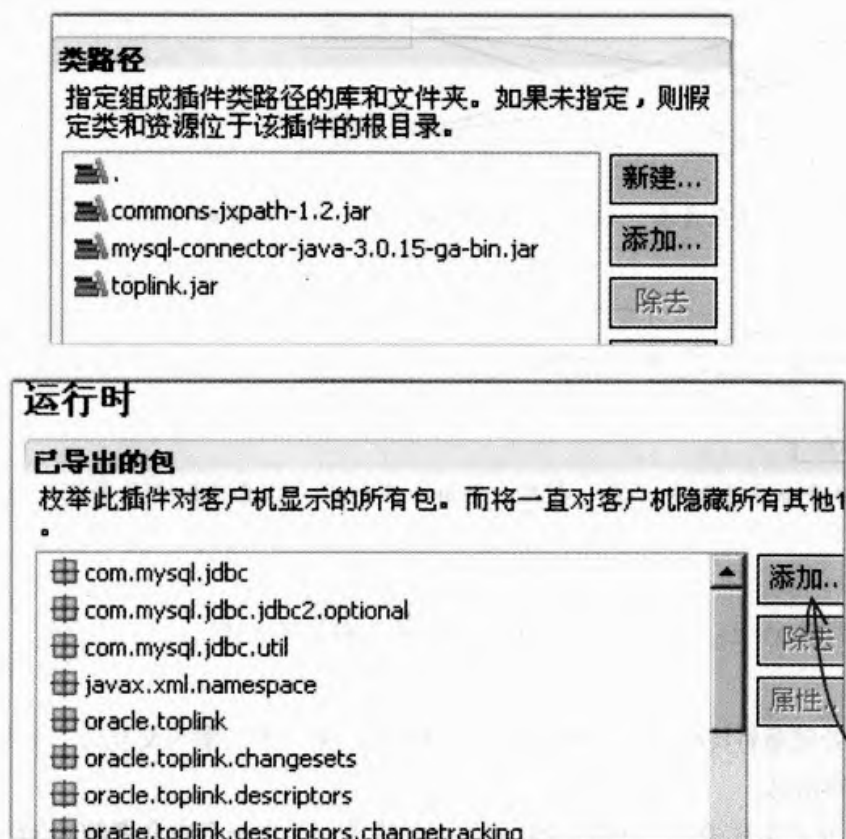


图23-10 创建thirdparty插件，导出所有第三方包

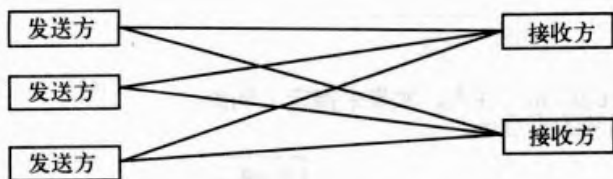
23.2.2 插件rcpdev.common.core和rcpdev.common.ui

rcpdev.common.core和rcpdev.common.ui插件包含了在程序中可能被重用的一些功能类、独立组件等内容。本节将对其中较重要的两个部分予以介绍，理解这些内容是理解示例程序代码的基础。

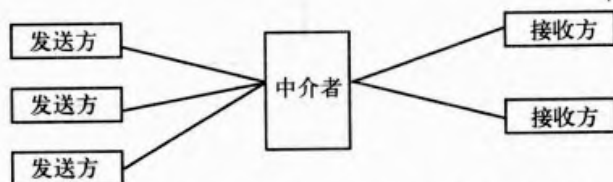
23.2.2.1 中介者Mediator

在common.core插件中包含了一个中介者（Mediator）模式的实现，这是为了方便插件中各个组件交互而设计的（rcpdev.common.core.mediator.*）。

不同的组件交互时，为了减少各部分之间的耦合性，可以使用观察者（Observer）模式，但当需要交互的组件很多时，如果每个组件都需要向其他各组件注册观察者，在修改或删除一个组件时，其他所有引用到该组件的组件都需要改动，这就会导致比较大的变动，对程序的稳定性不利，这时可以考虑采用中介者模式。使用该模式时，所有发送通知的组件和希望监听通知的组件都只需要向中介者注册并和它交互，中介者会监听组件发送出的消息并将它们转发到监听通知的组件那里，图23-11显示了普通的观察者模式与中介者模式的对比。



普通的观察者模式：点对点连接，如果一个部件要退出，它的所有合作者都要改变



使用中介者：所有部件都连接到中介者，如果一个部件要退出，只要通知中介者就可以了



图 23-11 普通的观察者模式与中介者模式的对比

使用中介者最大的好处就是将传统的“点对点”的同事角色解耦，所有参与交互的组件不需要了解除中介外其他任何组件的信息。

示例插件中包含的中介者实现采用了单例（Singleton）模式来提供一个中介者的实例，它基于 `PropertyChangeListener` 接口，所有采用这个接口进行通信的组件都可以方便地转为使用中介者。

代码如下所示(代码见光盘 `rcpdev.common.core.mediator.Mediator.java`)。

```
public class Mediator implements PropertyChangeListener {  
  
    private static Mediator instance;  
    public static Mediator getInstance() {  
        if(instance == null)  
            instance = new Mediator();  
        return instance;  
    }  
  
    private Mediator() {}  
  
    .....  
    public void addPropertyChangeListener(PropertyChangeListener listener) {  
        getDelegate().addPropertyChangeListener(listener);  
    }  
    public void removePropertyChangeListener(PropertyChangeListener listener) {  
        getDelegate().removePropertyChangeListener(listener);  
    }  
  
    public void propertyChange(PropertyChangeEvent evt) {  
        firePropertyChange(evt);  
    }  
}
```

代码中标记说明如下：



中介者本身实现了 *PropertyChangeListener* 接口，可以作为监听器被添加到发送消息的组件。



单例模式的实现代码。



当中介者收到消息时不会做任何处理，而是直接转发到所有正在监听中介者的监听器。

23.2.2.2 DataBinding 框架

rcpdev.common.ui 插件中包含了一个 DataBinding 的框架，DataBinding 的基本思想仍然是 MVC 模式，它将界面上控件的内容与后台数据模型的属性绑定起来，当用户操作控件时，数据模型的内容会即时更新，而开发者编写代码修改数据模型时，界面上控件的值也会随着刷新。这样，开发者在需要取得用户输入数据时就可以直接访问数据模型，而不需要遍历所有控件逐个取得它们的内容；而在程序中需要改变控件的值或实现控件联动（改变了 A 控件的值，B 控件要随之变化）时，也只需要改变数据模型中相应的值就可以了。示意图如图 23-12 所示。

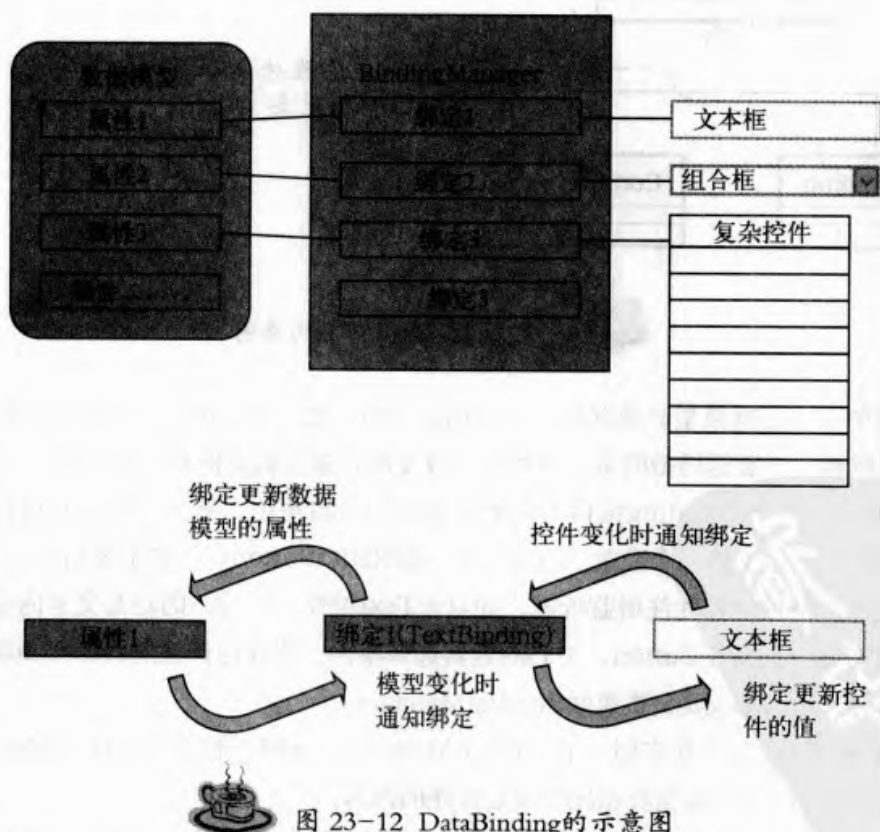


图 23-12 DataBinding 的示意图

在 Eclipse 3.2 版中，JFace 首度引入 DataBinding 的概念，但或许是开发尚未成熟，该版本中并未包含完整的 API。在本示例中自行实现了一套简易的 DataBinding 框架，这样做的主要目的是帮助读者了解 DataBinding 功能的基本原理，万变不离其宗，无论 JFace 在后续版本中推出的是怎样的 API，

只要了解了其原理，都可以很快上手。该框架的类图如图23-13所示。

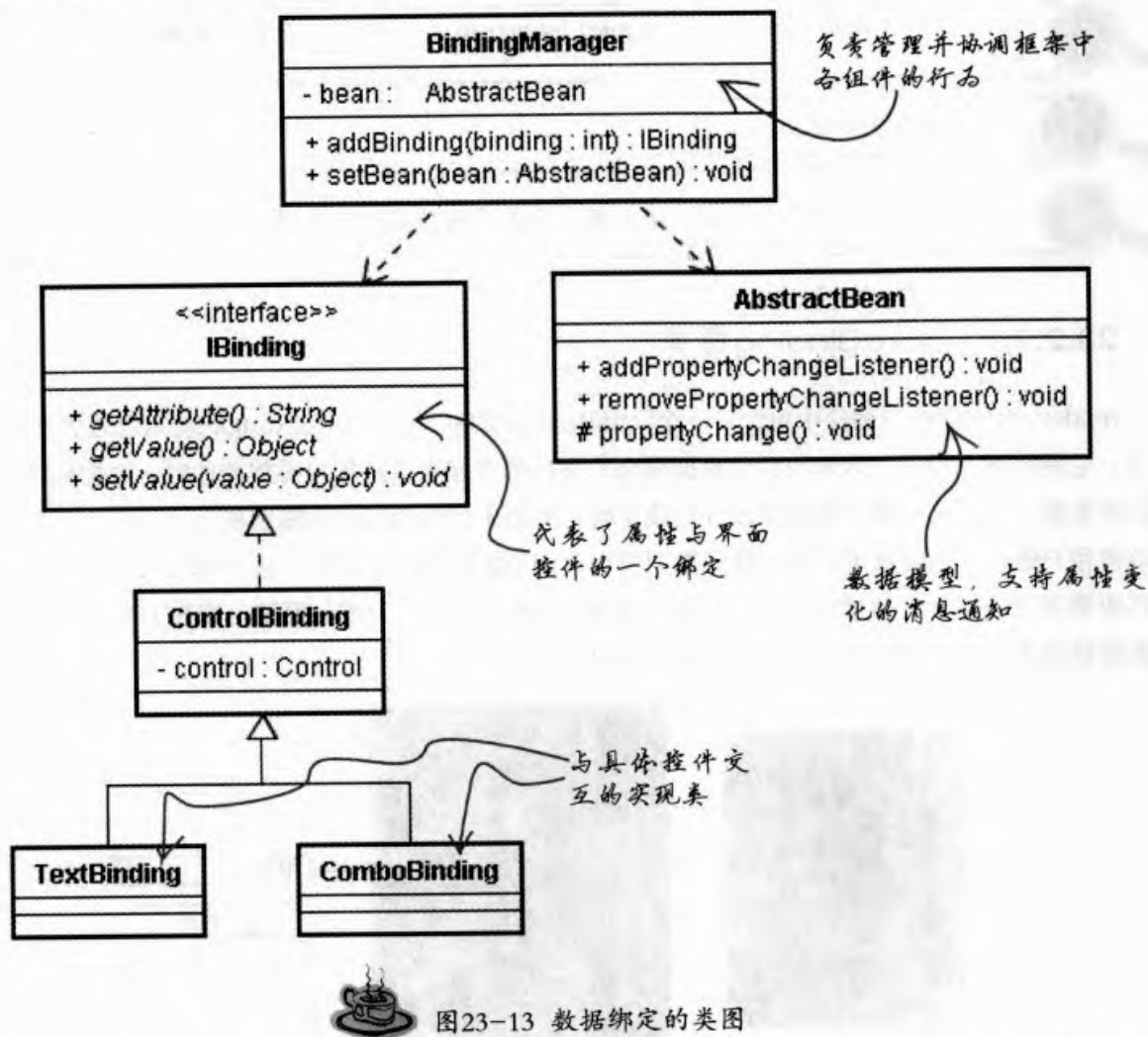


图23-13 数据绑定的类图

在这套框架中，数据模型必须继承自 **AbstractBean** 父类，该父类允许其他组件向数据模型上添加或移除属性观察者，当数据模型的某个属性发生改变时，模型就会向观察者发出通知。

BindingManager 和 **IBinding** 接口共同承担起控制器的角色。每一个 **IBinding** 对象对应着数据模型中的一个属性和界面上的一个或数个控件（如一组 **RadioButton**），它主要负责以下两部分工作。

★根据控件的特性向控件注册监听器，如对于 **Text** 控件，所关心的是其文本内容，因此应该注册 **Modify** 事件监听器，而对于 **Button**，关心的是其选择事件，因此应该注册 **Selection** 事件监听器等，当控件内容改变时，**IBinding** 要负责通知 **BindingManager**。

★当数据模型的内容改变时，**BindingManager** 会把属性的新取值通知对应这个属性的 **IBinding**，这时 **IBinding** 也要负责相应的设定控件的内容。

而 **BindingManager** 则负责在 **IBinding** 和数据模型之间进行沟通，当 **IBinding** 发出“控件内容已改变”的通知时，**BindingManager** 会相应地修改数据模型的属性；而数据模型的某一个属性变化时，**BindingManager** 也要找出属性所对应的 **IBinding** 对象并通知它这一变化，整个流程如图23-14所示。

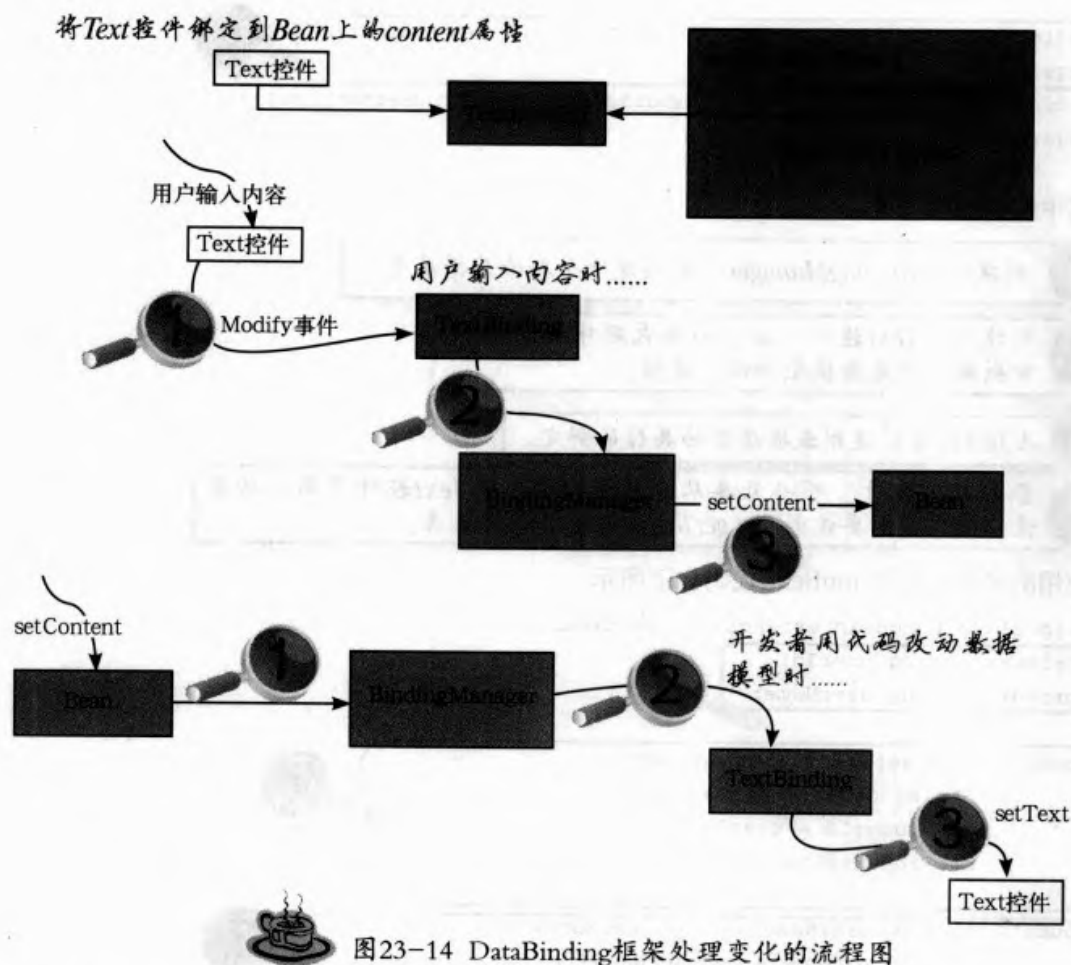


图23-14 DataBinding框架处理变化的流程图

使用该框架时，在创建了界面上的控件后，需要针对数据模型中每一个属性创建一个IBinding的对象，然后将它们绑定到对应的控件，根据要求不同，这些IBinding对象可能针对一个控件（如图23-14中的Text控件），也可能针对数个控件（如一组Radio Button等）。创建了这些属性到控件的绑定后，需要创建一个BindingManager对象来管理它们，随后可以调用BindingManager.addBinding方法向BindingManager添加绑定。下面的代码演示了如何在程序中使用该框架进行数据绑定（代码见光盘\book.ch23.rcpdev.common.ui.databinding.demo.DemoShell.java）。

```
final BindingManager manager = new BindingManager(new DemoBean());
.....
Text userIdText = new Text(shell, SWT.NONE);
Text userNameText = new Text(shell, SWT.NONE);
Button button = new Button(shell, SWT.NONE);
button.setText("Print Content");
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        DemoBean bean = (DemoBean)manager.getBean();
        System.out.println(bean.getUserId());
        System.out.println(bean.getUserName());
    }
});
```



```

IBinding idBinding = new TextBinding("userId", userIdText);
manager.addBinding(idBinding);
IBinding nameBinding = new TextBinding("userName", userNameText);
manager.addBinding(nameBinding);
.....

```

代码中标记说明如下:

创建一个`BindingManager`，并创建一个新的数据模型。

创建两个`Text`控件，在下面的代码中会将它们分别绑定到数据模型的两个属性。

为`Text`控件创建到数据模型的属性的绑定。

当单击按钮时，可以直接从数据模型中取得`Text`控件中输入的数值，而不再需要调用`Text.getText`方法取得这一数值。

所使用的数据模型`DemoBean`代码如下所示。

```

public class DemoBean extends AbstractBean {
    private String userId;
    private String userName;
    .....
    public void setUserId(String userId) {
        String oldId = getUserId();
        this.userId = userId;
        firePropertyChange("userId", oldId, userId);
    }
    public void setUserName(String userName) {
        String oldName = getUserName();
        this.userName = userName;
        firePropertyChange("userName", oldName, userName);
    }
}

```

代码中标记说明如下:

数据模型中定义了两个属性，在创建绑定时，会用这些属性的名称来引用它们。

在调用数据模型的`set`方法时，模型需要发送属性变化的通知消息出去。

使用了数据绑定功能后，开发者不再需要逐个处理界面上繁多的控件，而只需要直接访问数据模型就可以达到目的。在本示例程序中，所有界面上控件的管理都使用数据绑定功能完成。

框架中已经提供了用于部分常见控件的数据绑定实现类，如用于`Text`控件的`TextBinding`和用于`Label`控件的`LabelBinding`等，如果开发者需要实现对其他控件的支持，可以自行实现`IBinding`接口完成这一工作。

23.2.2.3 JavaBean

JavaBean是独立的、可重用的代码组件，在GUI程序设计中，JavaBean通常指一个可以重用的

界面组件，示例程序中包含了CalendarComposite和DateTextField两个JavaBean组件。

CalendarComposite是一个日历组件，它实现了基本的选择日期的功能，如图23-15所示。

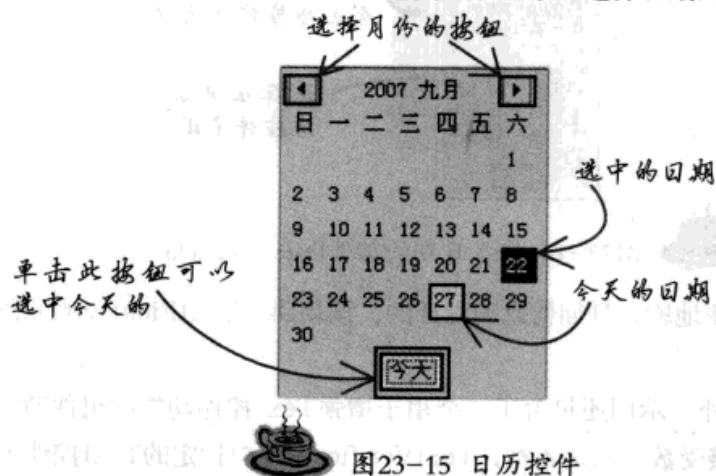


图23-15 日历控件

CalendarComposite可以被放置在任意的SWT窗口中使用，调用它的getDate方法可以得到用户当前选中的日期，当用户选择的日期改变时，CalendarComposite会发送一个“日期属性变化”的事件，如果程序希望在用户选择的日期改变时得到通知，可以监听这个事件。下面的代码简单演示了如何在一个Shell中使用CalendarComposite（代码见光盘：\book.ch23.rcpdev.common.ui.javabeans.demo.CalendarDemo.java）。

```
CalendarComposite calendar = new CalendarComposite(shell, SWT.NONE);

calendar.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent evt) {
        if (CalendarComposite.CALENDAR_DATE.equals(evt
            .getPropertyName())) {
            System.out.println(evt.getNewValue());
        }
    }
});
```

代码中标记说明如下：

在父容器中创建一个CalendarComposite对象。

从该对象发出的CALENDAR_DATE属性变化事件说明用户选择的日期已变，调用事件的getNewValue方法可以得到最新选中的日期。

而DateTextField则是一个允许用户输入日期的控件，它也重用了CalendarComposite。

DateTextField由一个只读的文本框和一个ARROW样式的Button组成。单击下拉列表Button后，会弹出一个包含日历控件的Shell，用户可以在日历控件里选择日期，被选中的日期会回填到文本框中，如图23-16所示。

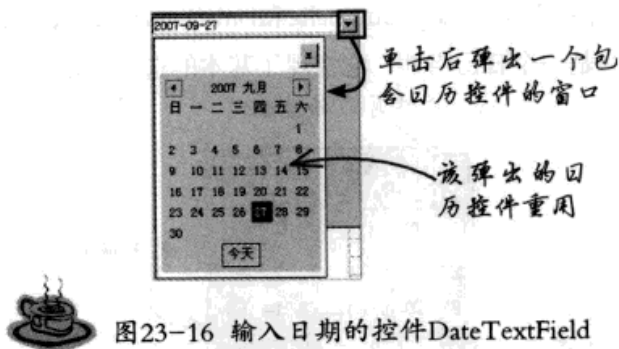


图23-16 输入日期的控件DateTextField

这个控件是为了让用户方便地输入日期信息而设计的，调用DateTextField.getDate方法就可以得到用户选中的日期。

除这两个界面相关的组件外，示例还包含了一个用于增强Text控件功能的组件TextVerifier，使用它可以创建长度限定、只接受数字的文本框，TextVerifier通过向指定的Text控件上添加Verify事件的监听器来完成这项工作，读者可以自行参考其实现代码（代码见光盘：book.ch23.rcpdev.common.ui.javabeans.text.TextVerifier.java）。

下面的三行代码分别为三个Text控件创建了TextVerifier，第一个Text控件被限定为只能输入10个字符，第二个控件可以输入20个数字字符，在第三个控件中可以输入浮点数，最长为30个字符。

```
new TextVerifier(text_1, 10, TextVerifier.TYPE_NORMAL);
new TextVerifier(text_2, 20, TextVerifier.TYPE_INTEGER);
new TextVerifier(text_3, 30, TextVerifier.TYPE_DOUBLE);
```

文本框

文本框类型：

普通
整数
浮点数

23.2.3 插件rcpdev.todo.core

使用UML设计工具的导出代码功能，可以方便地将画好的UML类图转化为模型代码（rcpdev.todo.core.model.*）。在Eclipse工作台中使用“源代码”菜单中的“生成Getter和Setter”功能，可以方便地为属性生成存取方法，除了UML类图中声明的属性外，示例还为模型中各个对象实现了clone方法，并重载了equals和hashCode方法。

数据模型代码中，与“日期过滤”这个核心业务逻辑相关的有以下几个重要方法。

- ★TodoItem.fitDate()/TodoSeries.fitDate()方法，外部代码调用该方法来执行日期过滤。
- ★SeriesInfo.fitDate()，在这里实现了前述SeriesInfo责任链的模式。
- ★DayInfo.fit()/WeekInfo.fit()/MonthInfo.fit()方法，这些方法中包含了具体的日期过滤逻辑。

在core插件中，还要声明一个用于访问持久化服务的接口以及一个扩展点。这样其他插件就可以通过扩展这个扩展点，并提供一个接口的实现类来为core插件提供数据模型的持久化服务了，这个扩展点被命名为facade(全名rcpdev.todo.core.facade)，它所对应的接口类是ITodoFacade，其代码如下所示，facade如图23-17所示。

```

public interface IToDoFacade {
    public void init() throws TodoFacadeException;
    public TodoItem addTodoItem(TodoItem item) throws TodoFacadeException;
    public TodoItem updateTodoItem(TodoItem item) throws TodoFacadeException;
    public boolean removeTodoItem(TodoItem item) throws TodoFacadeException;
    public List<TodoItem> findTodoItemByDate(Date date) throws
    TodoFacadeException;
}

```

持久化服务提供的基本接口

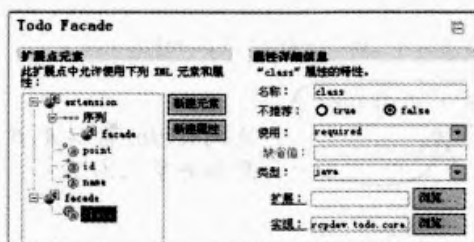


图23-17 扩展点rcpdev.todo.common.core.facade

工厂类TodoFacadeFactory负责从扩展中读取并实例化ITodoFacade的实现，这个类使用了Singleton模式，在其他插件中可以访问它来使用持久化服务。其代码如下所示。

```

public class TodoFacadeFactory {
    private static TodoFacadeFactory instance;
    public static final TodoFacadeFactory getInstance() {
        if (instance == null) {
            instance = new TodoFacadeFactory();
        }
        return instance;
    }
    .....
    private TodoFacadeFactory() {
        try {
            IExtension extension = Platform.getExtensionRegistry()
                .getExtensionPoint("rcpdev.todo.core.facade")
                .getExtensions()[0];
            facade = (ITodoFacade)
            extension.getConfigurationElements()[0]
                .createExecutableExtension("class");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

代码中标记说明如下：

Singleton模式的实现代码。

通过读扩展点的信息，取得ITodoFacade的实现类并据此创建对象。

在开发示例插件时,采取的是先开发UI部分,后开发持久化服务的顺序,然而没有持久化服务的支持,无法全面测试UI的功能,因此在core插件中为ITodoFacade提供了一个“伪持久化”的实现TodoFacadeMock。这个实现简单地将所有添加的TodoItem保存在一个Vector对象中,使用这个实现,开发者可以在没有持久化模块的情况下完成UI部分的开发与测试,如果有两部分开发人员的时候,还可以实现并行开发。等到真正的持久化模块编写完成后,只需要将TodoFacadeMock所对应的扩展删除,再为真正的持久化实现类提供一个扩展,就可以轻松的将两部分结合起来,这也体现了模块化编程的好处。代码如下所示。

```
public class TodoFacadeMock implements ITodoFacade {
    private Vector<TodoItem> storage;
    .....
}
```

使用Vector存储用户添加的事项内容

23.2.4 插件rcpdev.todo.ui

UI是整个示例程序的重点所在,本示例程序中的UI由两个视图和一个编辑器构成,如图23-18所示。



图23-18 To-Dos插件程序的界面

界面开发相关的技术问题，包括视图和编辑器的设计，国际化程序等内容，在本书前面的部分已经有涉及，此处不再赘述。本节将以用户操作所对应的代码调用过程为主线，介绍UI部分的代码架构，这样可以帮助读者理清思路，在阅读实例代码时也会更为清晰。

由用户通过工具栏或菜单操作触发的功能，基本上是以两层架构划分的，如图23-19所示，其说明如下所示。

1. 操作和各种对话框，这一层负责与用户发生交互。
2. Job会通过后台线程访问持久化服务，并把得到的结果返回给UI线程。关于Job的使用可见“使用Eclipse Job API”一节内容。

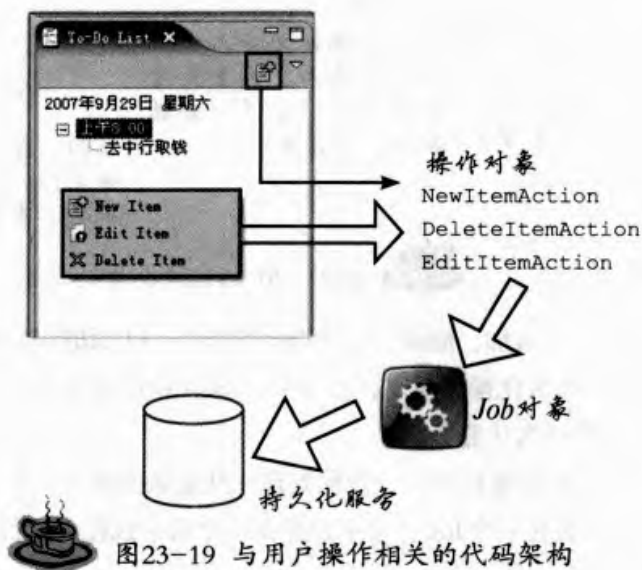


图23-19 与用户操作相关的代码架构

在To-Do List的界面上，用户可以从工具栏按钮新建一个项目，也可以从右键菜单完成新建、编辑、删除等工作。新建一个项目时，NewItemAction会使用NewItemWizard来生成创建新项目的向导对话框，在该向导的performFinish方法中，启动了CreateToDoItemJob，而后者最终调用了ITodoFacade的addToDoItem方法将项目添加到持久化服务中。

编辑与删除操作的流程与此大致相似，由于这两个操作都需要用户确认操作对象，因此EditItemAction与DeleteItemAction都继承自ItemOperationAction父类，这个父类在运行时会显示一个对话框，要求用户选择操作整个系列还是单独一项的内容，用户选择后，DeleteToDoItemJob会直接调用ITodoFacade.removeToDoItem方法，而EditToDoItemJob则会生成一个ToDoItemEditorInput对象，并启动编辑器来编辑事项内容。

在编辑器ToDoItemEditor的doSave方法中，启动了SaveToDoItemJob，后者最终访问到ITodoFacade.updateToDoItem方法，完成整个编辑过程。

以上描述了系统中处理用户通过工具栏/菜单直接触发的操作，如果涉及不同组件之间的交互，如在日历视图中改变了选择的日期，To-Do List视图会随之刷新这样的情况，则是通过中介者处理的。

在示例插件中，共有三个部件通过中介者进行交互，其中日历视图和持久化服务作为消息的发送方出现，而待办事项视图则作为消息的接收方出现，如图23-20所示。

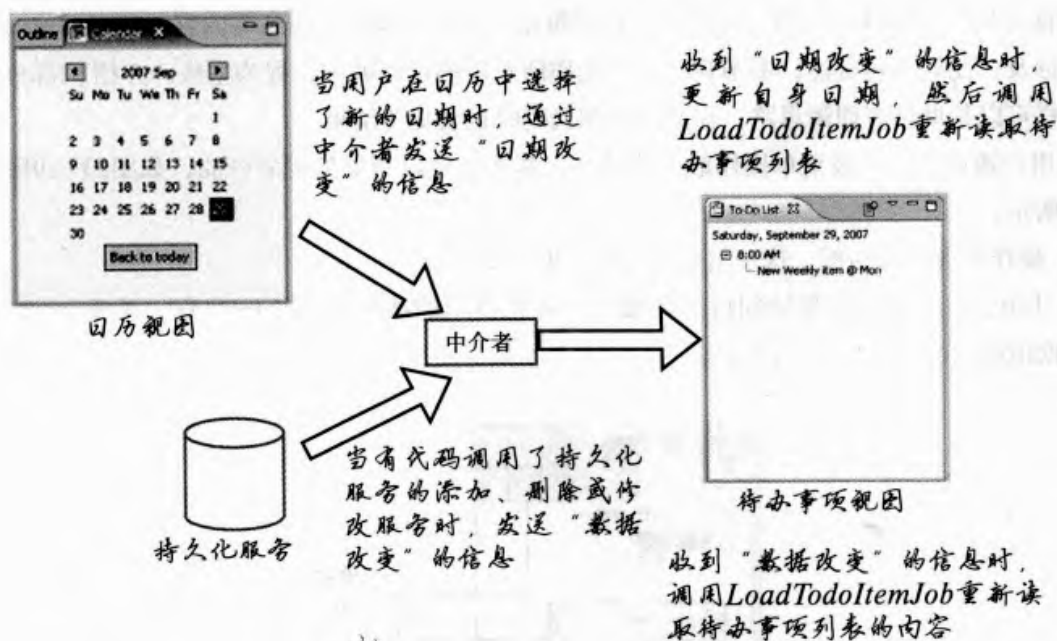


图23-20 通过中介者的交互

在示例程序中，当用户添加、删除待办事项时都需要从UI访问持久化服务，为了保证用户的界面操作不会被比较耗时的持久化操作阻塞，在todo.ui插件中广泛使用了Eclipse Job API(org.eclipse.core.runtime.jobs)来调用持久化服务。




编写插件程序时，如果需要利用后台线程处理一些复杂的操作，都可以考虑使用这套API，它允许开发者将可执行代码包装在一个Job对象中，并在一个后台线程中执行这些代码。代码如下所示。

```
Job job = new Job("My First Job") {
    protected IStatus run(IProgressMonitor monitor) {
        System.out.println("Hello World (from a background job)");

        return Status.OK_STATUS;
    }
};

job.setPriority(Job.SHORT);
job.schedule();
```

代码中标记说明如下：

-  创建一个Job类并在它的run方法中编写需要在后台线程中执行的代码。
-  为Job设置优先级。
-  通知Eclipse的工作管理器该Job对象已经准备好被执行。工作管理器会根据调度算法在恰当的时间启动一个线程执行这项工作。

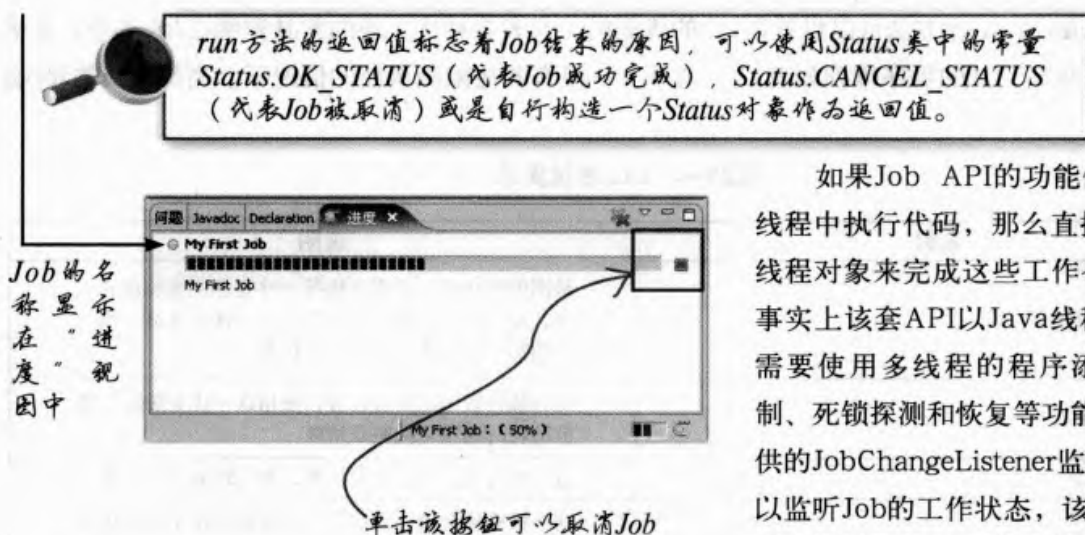


图23-21 使用“进度”视图监测Job的状态

Job对象被创建出来后，可能立刻被执行，也可能由于某些执行条件尚未满足而处于等待状态，这些状态由state属性表示，对应的状态变化图如图23-22所示。

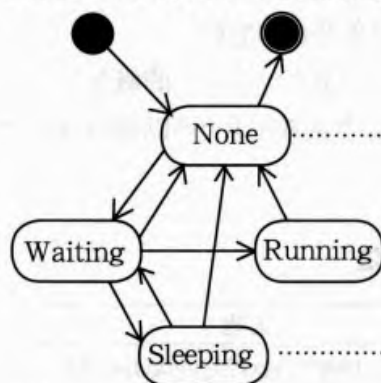


图23-22 Job的状态变化图

表23-3列出了对各个状态及变化的说明。

表23-3 Job的状态含义、变化与说明

状态名	说明
Job.NONE	新创建、尚未被调度的Job，执行完毕的Job或被cancel的Job均处于这一状态
Job.WAITING	这个状态代表Job已经被加入调度队列等待执行，调用Job.schedule方法可以将处于NONE状态的Job转为这一状态，调用cancel方法则会使该状态的Job回到NONE状态，当Job开始执行时，就会变到RUNNING状态
Job.RUNNING	该状态代表Job正在运行中，当Job运行完成后，会进入NONE状态
Job.SLEEPING	该状态代表Job已经被挂起，挂起的Job不能被执行，调用Job.sleep可以将处于NONE或WAITING的Job转为这一状态，调用Job.wakeUp可以使Job回到WAITING状态，而Job.cancel则会使Job回到NONE状态

调用Job.setPriority方法可以设置该Job的优先级，代表各种优先级的常量声明在Job类中，其名称及含义按照从高到低的顺序排列如表23-4所示（在没有其他限制条件的情况下，高优先级的Job会先被执行）。

表23-4 Job的优先级

名称	说明
Job.INTERACTIVE	最高的优先级，适用于执行一些要求快速响应的任务，如处理用户界面输入等，这一优先级适合执行时间短，要求响应速度快的任务
Job.SHORT	适合执行较短的后台任务，使用这一优先级时，任务的耗时通常应控制在秒级
Job.LONG	适合执行长时间的后台任务，如比较复杂的计算等
Job.BUILD	这一级别通常用来安排担任总结或扫尾工作的任务，它会在其他关键性任务都执行完毕后开始执行
Job.DECORATE	最低的优先级，这一级别通常用来执行非关键性的任务，这类任务即使没有完成也不会影响程序正常执行

Job的调度与执行是由工作管理器(Job Manager)完成的，可以调用Platform.getJobManager得到当前使用的工作管理器并通过IJobManager接口访问它，也可以通过Job.getJobManager方法得到某一个Job对象所属的管理器。通过工作管理器可以查找到Job对象并操作它们。

表23-5列出了工作管理器中常用的方法，其中涉及到了“家族(family)”的概念。在工作管理器中，Job对象是被分组管理的，每个组称为一个家族，同属一个家族的Job可以同时被取消、挂起或恢复。

表23-5 工作管理器的常用方法

方法名	说明
suspend()	挂起管理器，所有等待执行的Job都会被暂停，已经在运行的Job则不受影响
resume()	如果管理器处在挂起状态则恢复管理器，被暂停的Job将恢复到暂停之前的状态
find(Object family)	找出所有属于family代表的家族的Job
sleep(Object family)	挂起所有属于family家族的Job，该操作等同于用find操作找出该家族的Job，并同时调用它们的sleep方法（下同）
wakeup(Object family)	唤醒所有属于family家族的Job
cancel(Object family)	取消所有属于family家族的Job

开发者调用工作管理器中与家族相关的方法，如find,sleep等时，工作管理器会调用Job类的belongsTo(Object family)方法来判定某个Job对象是否属于给定的家族，因此Job对象应该重载Job.belongsTo方法来实现家族的分组逻辑，下面给出一个例子，代码如下所示。

```
public class FamilyJob extends Job {  
    private String lastName;
```

```
public FamilyJob(String firstName, String lastName) {
    super(firstName + " " + lastName);
    this.lastName = lastName;
}
```

```
public boolean belongsTo(Object family) {
    return lastName.equals(family);
}
.....
```

代码中标记说明如下:

1 创建Job时, 要为Job提供一个firstName和一个lastName。

2 Job会根据family对象是否等于自己的lastName来判定是否属于给定的family。

下面的代码演示了如何通过工作管理器访问同属一个家族的Job对象并操作它们。

```
new FamilyMember("Jack", "Jones").schedule();
new FamilyMember("Lucy", "Jones").schedule();

new FamilyMember("Debbie", "Harper").schedule();

IJobManager manager = Platform.getJobManager();

Job[] jones = manager.find("Jones");
manager.sleep("Jones");
manager.wakeUp("Jones");
manager.cancel("Jones");
```

代码中标记说明如下:

1 创建两个lastName为“Jones”的Job, 根据FamilyJob的belongsTo方法的判断逻辑, 它们属于同一个家族。

2 第三个Job与前面两个属于不同家族。

2 使用find等方法操作整个家族的Job。

可见, 工作管理器并不关心用户传入的代表家族信息的对象的具体内容, 区分工作完全由Job自身完成。

如果用户希望了解后台有哪些线程正在执行, 执行进度等信息, 就要用到Job API中与界面显示相关的信息。从图23-21中可以看到, 如果当前有Job正在运行, 工作台窗口的右下角会有一个进度指示器出现, 在“进度”视图中也有对应的内容, 如果开发者希望更明确地提醒用户“运行了一个Job”, 可以通过把Job的类型设置为“用户型”来达到这一目的, 如下面代码所示。




```
job.setUser(true);
```

用户型Job在运行时，会弹出如图23-23所示的提示对话框，用户可以等待Job完成，对话框消失，也可以选择“Run in Background”关掉对话框，Job继续在后台运行。

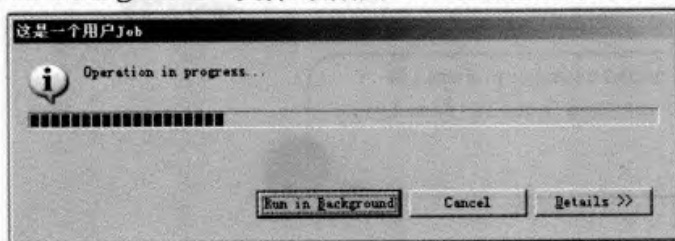


图23-23 用户型Job运行时的提示对话框

为了在“进度”视图中及时向用户反映Job的进度，需要在Job的run方法中向工作管理器通报进度信息，run方法的参数中提供了一个IProgressMonitor的对象，通过操作它可以完成这一任务，而当用户在界面上取消一个Job时，信息也是通过这个对象传递给Job的。IProgressManager提供的API与第7章中讲述的进度指示器很相像，下面的代码建立了一个Job，它每一秒钟将进度条推进十分之一，用户可以中途取消执行，如图23-24所示。

```
protected IStatus run(IProgressMonitor monitor) {
    try {
        monitor.beginTask("用户Job的任务", 100);

        for(int i=0; i<10; i++){
            if(monitor.isCanceled())
                return Status.CANCEL_STATUS;

            monitor.worked(10);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return Status.OK_STATUS;
}
```

调用beginTask时指定的名称显示在这里

代码中标记说明如下：

1 设定总任务量为100。

2 如果用户单击了界面上的取消按钮，monitor.isCanceled方法会返回true，这时Job可以直接返回一个STATUS_CANCEL状态。

3 每次完成量为10的工作。

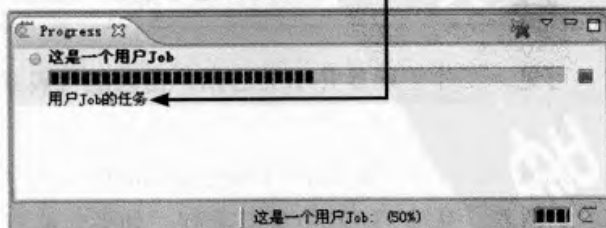


图23-24 Job执行时显示在“进度”视图中

※ 注意: ※

设置Job为用户型的工作必须在调用Job.schedule方法前完成, 如果Job已经开始运行了, 再去设置Job的类型是无效的。



最后, 将介绍Job API是如何对多线程进行同步控制及死锁探测的, 这也是Job API的最大亮点所在。

程序代码的运行需要各种资源的支持, 如文件、网络连接等都属于资源的范畴, 多数资源是非共享的, 这意味着两个线程不能同时操作它, 如A线程在写文件F, B线程如果也想写该文件, 就必须等待A线程的操作完成。在多线程环境中, 如果两个线程开始争夺非共享的资源, 则有可能出现死锁的情况, 例如两个线程A和B都需要资源X和Y以完成执行过程, 某一时刻, 线程A持有了资源X, 等待资源Y被释放; 而线程B则持有了Y, 等待X被释放, 这就构成了死锁, 如图23-25所示。

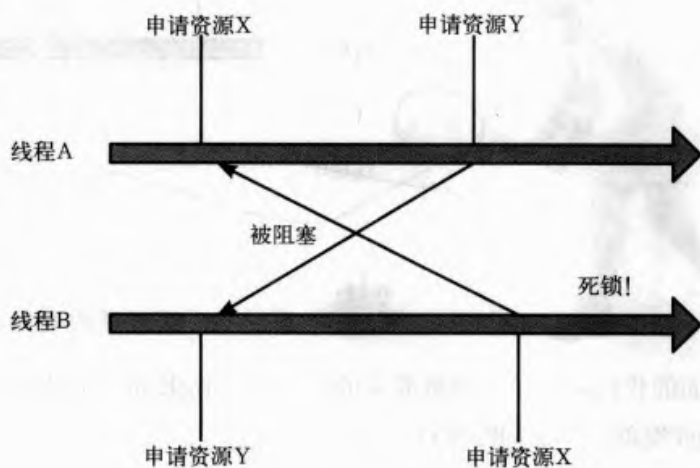


图23-25 死锁示意图

Job API通过ISchedulingRule接口管理资源和调度Job, 在生成Job的时候, 可以通过setRule方法为它指定一个ISchedulingRule对象, 工作管理器根据该对象提供的信息制定执行该Job对象的规则。如下面代码所示。

```
ISchedulingRule rule = .....;  
job.setRule(rule);
```

为了避免死锁情况的发生, 调度算法满足以下两个要求就可以了(充分条件)。

★如果两个Job所需要的资源有冲突, 它们不能同时运行。

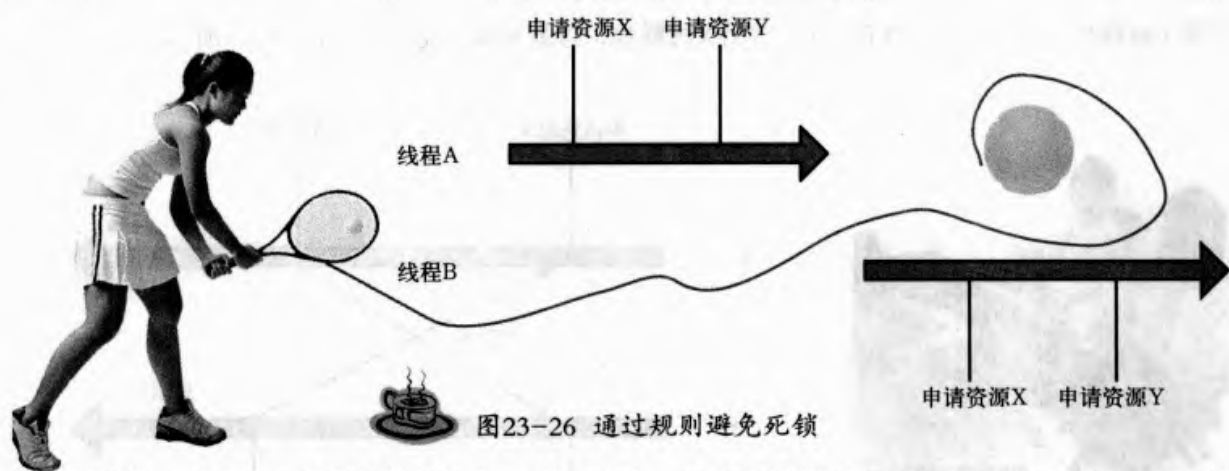
★Job取得所有它需要的资源后才开始执行。

而ISchedulingRule中声明的两个方法正是针对这两点而定的, 如下所示。

```
public interface ISchedulingRule {  
    public boolean isConflicting(ISchedulingRule rule);  
    public boolean contains(ISchedulingRule rule);  
}
```


isConflicting方法用来判定两个Job所持有的规则是否冲突，有冲突的Job是不能同时运行的；而contains方法则用来判定规则之间的包含关系，只有当某个资源所对应的规则被Job的规则所包含，Job才能申请并使用该资源。在申请资源前，Job应该调用IJobManager.beginRule(ISchedulingRule resourceRule, ...)方法来请求工作管理器检查资源对应的规则是否包含在当前Job的规则中，而当使用资源结束后，也应该调用IJobManager.endRule(ISchedulingRule resourceRule)来表明使用结束。

仍以两个线程申请两个资源的情况为例，由于Job A需要资源X和资源Y，那么Job A的规则中会包含资源X和资源Y对应的规则；同理，Job B的规则中也包含这些内容。因为X和Y都是非共享资源，所以A和B的规则必定是冲突的，因此如果A先开始执行，B会等到A执行完成后才开始，这就解决了死锁的问题，如图23-26所示。



下面的代码实现了一个最简单的ISchedulingRule，这个规则规定两个持有该规则的同一个实例的Job是冲突的，不能同时执行。

```
public class SingleRule implements ISchedulingRule {  
    public boolean contains(ISchedulingRule rule) {  
        return rule == this;  
    }  
    public boolean isConflicting(ISchedulingRule rule) {  
        return rule == this;  
    }  
}
```

该Rule只和自身冲突，只包含自身

在Job上使用如下代码所示的这条Rule。

```
ISchedulingRule singleRule = new SingleRule();  
Job a = .....;  
Job b = .....;  
a.setRule(singleRule);  
b.setRule(singleRule);  
a.schedule();  
b.schedule();
```

为两个Job使用SingleRule的同一个实例



这时, Job B就会一直等到Job A执行完毕后才开始执行。

23.2.5 插件rcpdev.todo.persistence

rcpdev.todo.persistence插件包含了基于TopLink、MySQL数据库的一个持久化实现,以及一个用于配置数据库连接的首选项页面,如图23-27所示。本书不会对O/RM技术的详细实现做讲解,如果读者对这一技术不是很熟悉,可以尝试将该插件更改成使用JDBC连接数据库的版本,这也有助于进一步体会示例程序的体系结构。

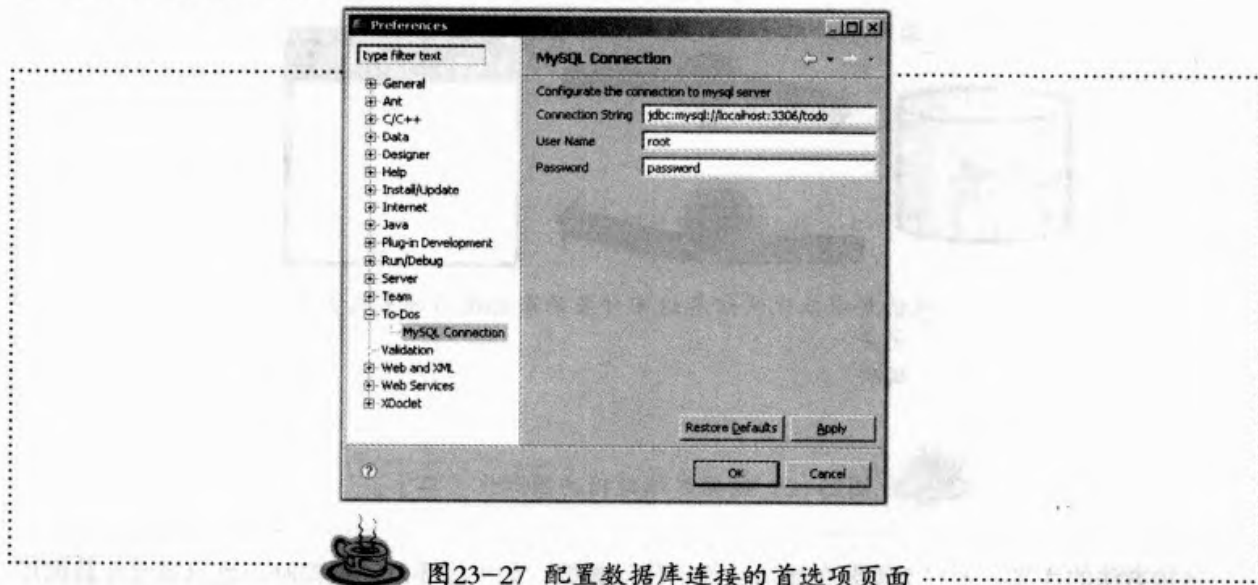


图23-27 配置数据库连接的首选项页面

关于如何国际化插件的UI,在第19章中已经有详尽的描述,然而在实际开发中,国际化的工作并不是全部集中在UI一层的。有时可能需要对后台存储(通常是数据库存储)的数据进行国际化。

对存储在数据库中的数据进行国际化通常有以下两种方式。

1. 在数据库中存储数据的索引,将国际化的工作交给显示层完成,如图23-28所示。

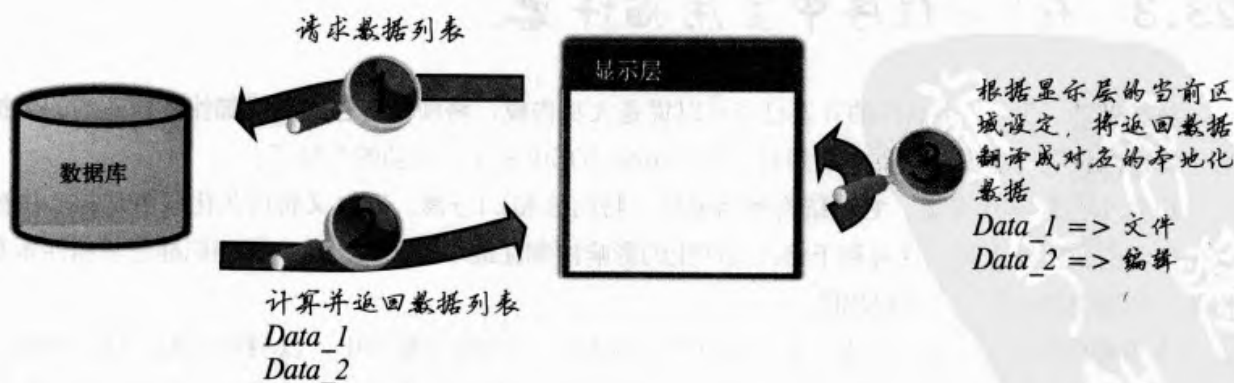


图23-28 数据库端数据的国际化(一)

这种方法的好处在于显示层统一处理所有国际化工作，这项工作对程序的其他部分完全透明。它的缺点在于显示层必须维护一套数据库可能返回的数值的最大集合。当数据库信息有变时，显示层也要随之更新，维护工作相对复杂。

这种方式适于在数据集合基本没有变动，而且数据库承担着取数据的主要计算工作的情况下使用。如果没有计算工作，那么整个数据集合可以直接硬编码在显示层中。

2. 在数据库中存储数据各种语言的版本，客户端要求数据时，连同所要求的语言版本一起提交，如图23-29所示。



这种方法的优势在于它不需要在显示层维护任何数据，如果需要改动数据时，也只需要在数据库中添加就可以了。其劣势在于一般无法重用显示层所提供的国际化框架，开发者需要自己实现一套与显示层兼容的国际化框架以进行数据库信息的本地化，工作较复杂而且不容易维护。

这种方式比较适合用来处理改动较频繁的数据集合。

23.3 在RCP程序中重用插件

到此为止，To-Dos插件的开发已经可以说是大功告成，将项目打包成功能部件并作适当的装饰后，就可以将它通过更新站点安装到自己的Eclipse IDE中来享受成功的喜悦了。

以这个插件本身来说，它的结构相当灵活。将内核和UI分离，同时又将持久化服务从核心中独立出来，当变化发生时，这有利于将改动产生的影响控制在最小范围内。这一评判标准是将插件本身作为一个完整的软件产品而提出的。

本节将尝试将To-Dos作为一个功能部件集成到已有的RCP程序中，以利用它的“待办事项列表”功能为RCP程序提供服务，这时对插件设计的要求也就有所不同，一般来说，在设计RCP程序时，更关注的是组成RCP的插件或功能部件的功能是否足够独立，插件之间不应该有过多地依赖，这样可以保证当某一部分出现问题时，程序只会丧失一部分功能，而不至于全盘崩溃，本节将以一个实例来展示如何将已有的插件结合起来构成RCP程序。

23.3.1 Contact插件介绍

为了管理联系人的信息，在To-Dos插件后，一个Contact插件被开发出来，它支持对联系人信息的添加、删除、修改和查找，同时也提供了它的RCP程序实现。其代码见rcpdev.contact.*插件。

Contact插件的界面如图23-30所示，其结构与To-Dos插件基本相似，也分为core, ui和persistence几个部分，这里将不再重复对Contact插件的具体实现加以介绍，读者可以参考前面对To-Dos插件的介绍自行阅读代码。

在Contact插件中，使用了对表格查看器排序的一些相关技巧，具体内容可以参照“查看器的排序”一节内容。

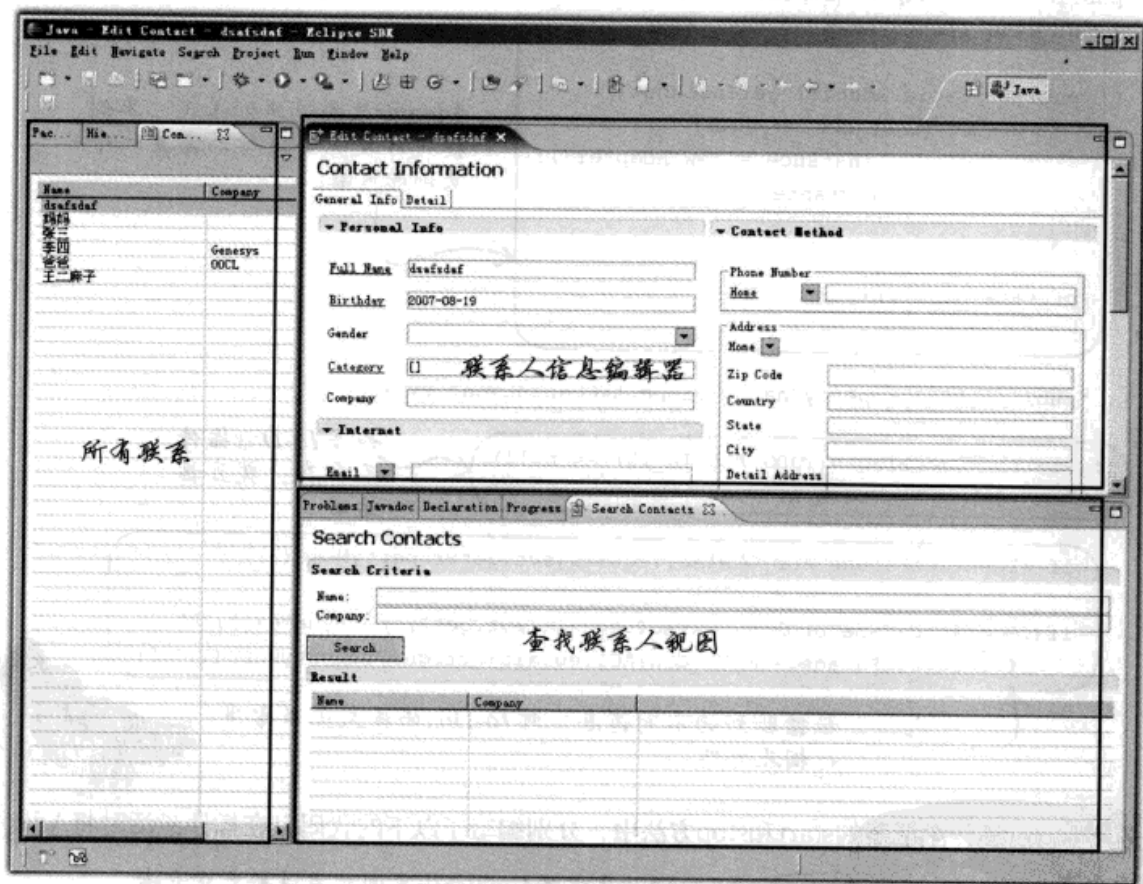


图23-30 在Eclipse中运行Contact插件的界面

在创建联系人时，可以为联系人添加生日信息，为了在联系人生日到来时得到通知，考虑使用To-Dos插件的事项列表功能记录生日，具体描述如下所示。

★当用户创建联系人时，创建一条标题为“xx(联系人姓名)的生日”的系列事项，将系列时间设定为“每年的x月x日(联系人生日)”，时间自动设为8:30分。

★当用户修改联系人姓名或生日时，自动更新这条系列的内容。

★当用户删除联系人时，删除该系列的内容。

首先要考虑当用户对联系人进行操作时，如何得到通知。在介绍To-Dos插件时提到，在调用持

久化服务对事项记录作修改时，持久化服务的实现会通过中介者发送“事项被修改”的通知；同样，在Contact插件中，如果用户通过界面插入、修改或删除联系人记录时，它的持久化服务也会通过中介者广播这一改变事件。因此只要向中介者注册，就可以得到这一信息了。而操作To-Dos插件的任务可以通过访问在rcpdev.todo.core中提供的持久化服务接口完成。

最后的问题是完成操作的代码应该放在哪里。出于对插件依赖性的考虑，不应该放在Contact插件内部，因为To-Dos插件如果被卸载了或出现了问题，Contact插件不应该受到影响，同样的道理，也不应该放在To-Dos插件内部。因此创建了一个新的插件rcpdev.adapter来完成这项任务，该插件中除激活器类外，只有一个Adapter类用于完成操作，其代码如下所示。

```
public class Adapter implements PropertyChangeListener {
    private static Adapter instance;

    public static Adapter getInstance() {
        if (instance == null)
            instance = new Adapter();
        return instance;
    }

    private Adapter() {
```

Adapter使用了单例模式，单例模式比较适合这种只需要一个实例的对象

```
    public void propertyChange(PropertyChangeEvent evt) {
```

```
        if (Platform.getPlugin(TODO_PLUGIN_ID) == null)
            return;
```

如果To-Dos插件不存在，就直接返回

```
        if (IContactFacade.ADD_CONTACT.equals(evt.getPropertyName())) {
        }
        if (IContactFacade.UPDATE_CONTACT.equals(evt.getPropertyName())) {}
        if (IContactFacade.REMOVE_CONTACT.equals(evt.getPropertyName())) {}
    }
}
```

根据收到的不同消息，对To-Dos的持久化服务进行相应操作

在Adapter插件激活器的start和stop方法中，分别编写了以下的代码以在插件激活时将Adapter注册到中介者。

```
Mediator.getInstance().addPropertyChangeListener(Adapter.getInstance());
```

```
Mediator.getInstance().removePropertyChangeListener(Adapter.getInstance());
```

在stop方法中将Adapter的实例从中介者删除

Adapter类必须在Contact插件启动之前就注册到中介者，这样才能及时捕捉到从Contact插件中发送出来的改变情况，使用org.eclipse.ui.startup扩展点可以达到这个效果。这个扩展定义在org.eclipse.ui插件中，它要求插件提供一个拥有默认构造函数的IStartup接口实现类，当工作台启动时，会初始化所有IStartup实现类，并调用IStartup.earlyStartup方法来执行初始化工作。下面的代码是Adapter插件中使用的启动类AdapterStartup。

```
public class AdapterStartup implements IStartup {
    public void earlyStartup() {}
}
```

由于初始化工作已经包含在插件激活类中，这个类的任务就只是激活插件Adapter，因此没有编写任何具体的代码

现在将Adapter插件和其他两个插件打包在一起，就可以得到一个拥有日程提醒功能的联系人管理器RCP程序了。图23-31显示了该RCP程序的结构。

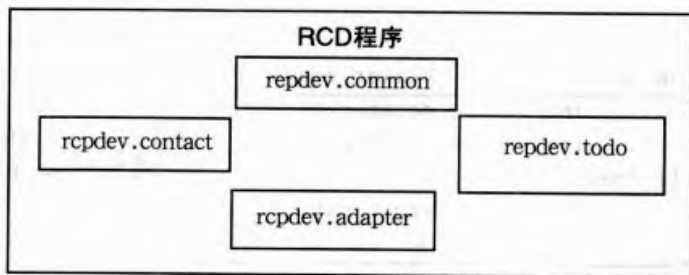


图23-31 RCP程序的结构

23.3.2 查看器的排序

在“所有联系人”列表中，为表格查看器添加了排序的功能，并允许用户通过菜单切换排序方式，本节将结合这一实现，对如何将查看器中的内容排序加以介绍。

在StructuredViewer类中(这个类是列表查看器、表格查看器、树查看器等常用查看器的父类)，排序的工作是由ViewerComparator来代理完成的。通过方法setViewerComparator可以设置作为代理的对象。在Eclipse 3.2之前的版本中，这一工作由ViewerSorter类来完成。自3.2版开始，JFace推荐使用ViewerComparator来代替它。

查看器刷新自己的内容时，如果它能够找到一个ViewerComparator实例，就会调用实例的sort(final Viewer viewer, Object[] elements)方法来得到一个排好序的对象数组，图23-32演示了这一系列的调用过程。

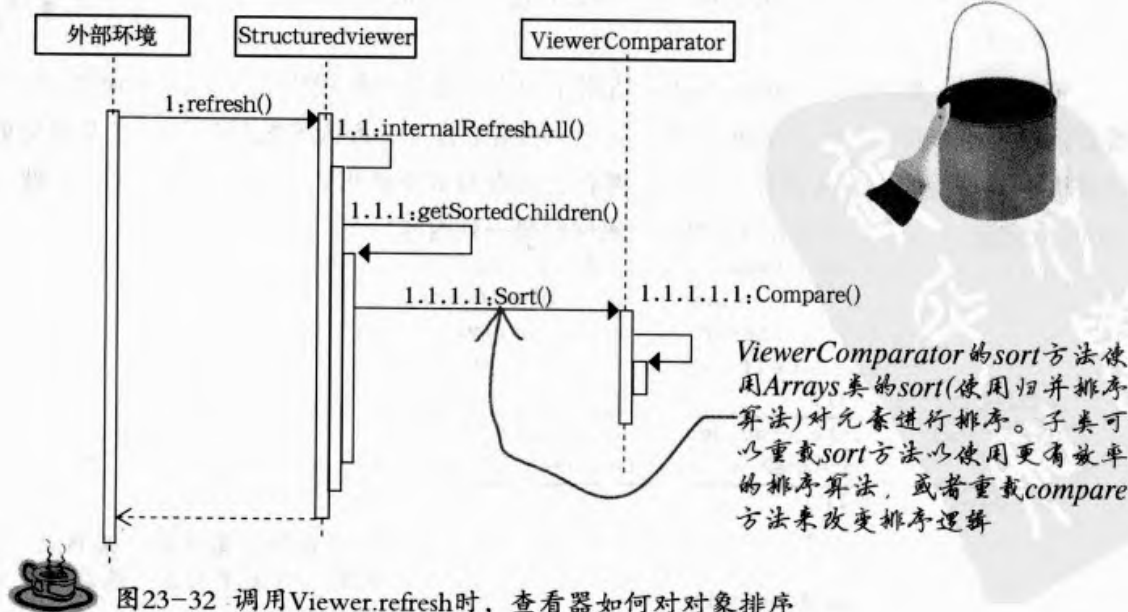


图23-32 调用Viewer.refresh时，查看器如何对对象排序



ViewerComparator不是一个抽象类，可以直接在程序中使用它，如下面代码所示。

```
tableViewer.setComparator(new ViewerComparator());
```

然而这种简单的使用也仅仅能提供最简单的功能。检查ViewerComparator的代码可以发现，它的compare方法逻辑如下所示。

```
public int compare(Viewer viewer, Object e1, Object e2) {
```

```
    int cat1 = category(e1);  
    int cat2 = category(e2);  
    if (cat1 != cat2) {  
        return cat1 - cat2;  
    }
```

category方法根据对象计算出一个类别号码。对象会首先按照类别号码排序

```
    .....  
    if (viewer == null || !(viewer instanceof ContentViewer)) {
```

```
        name1 = e1.toString();  
        name2 = e2.toString();
```

```
    } else {
```

```
        IBaseLabelProvider prov = ((ContentViewer) viewer).  
            .getLabelProvider();
```

```
        if (prov instanceof ILabelProvider) {  
            ILabelProvider lprov = (ILabelProvider) prov;  
            name1 = lprov.getText(e1);  
            name2 = lprov.getText(e2);
```

```
        } else {  
            name1 = e1.toString();  
            name2 = e2.toString();  
        }
```

```
    }
```

```
    .....  
    return getComparator().compare(name1, name2);
```

```
}
```

调用内置的Comparator对两个String对象作比较，默认的Comparator直接调用String的compareTo方法

如果所在查看器是一个使用IBaseLabelProvider的查看器(如列表查看器)，则以从标签提供方中取到的标签为比较目标，否则则以比较对象toString得到的结果为目标



可以看出，默认的compare方法只适用于列表查看器等使用IBaseLabelProvider提供内容的查看器，对表格查看器，它会根据对象toString的结果排序；而大多数情况下，开发者会更希望它根据表格中某一栏的内容来排序。如果需要在表格查看器中使用ViewerComparator，就一定要重载compare方法。下面是一个示例，它按照表格的第一列内容排序。

```
public int compare(Viewer viewer, Object e1, Object e2) {  
    assert viewer instanceof TableViewer;  
    TableViewer tViewer = (TableViewer) viewer;  
    ITableLabelProvider labelProvider = (ITableLabelProvider) tViewer.  
        getLabelProvider();
```

```
    String name1 = labelProvider.getColumnText(e1, 0);  
    String name2 = labelProvider.getColumnText(e2, 0);
```

```
    if (name1 == null) {  
        name1 = ""; //$NON-NLS-1$  
    }  
    if (name2 == null) {
```

从表格查看器对象中取得两个元素对应的第一行文字内容，然后据其进行排序

```

        name2 = ""; //$NON-NLS-1$
    }
    // use the comparator to compare the strings
    return getComparator().compare(name1, name2);
}

```

上面的代码已经清楚地演示了如何使用ViewerComparator对表格进行静态排序——这样排序的结果只和查看器的内容有关，而与用户操作无关。然而在使用表格控件时，更常见的是动态排序的问题，用户单击表格的某一栏头部，要求表格内容按照这一栏排序。进一步的要求是，单击一次时，表格内容以升序排列，再单击一次，则以降序排列；或者要求实现按多列排序(先按A列排序，再按B列排序)等。

在实例程序的ContactView(代码见光盘：\book.ch23.rcpdev.contactmgr.ui.views.contact)中，分别提供了单列排序和多列排序两种排序方式。可以通过视图菜单切换它们，如图23-33所示。



图23-33 ContactView提供的排序方法选择

单列排序：单击某一栏时，内容会按照这一栏自上向下升序排列。如果连续单击某一栏，则内容会按照“升序→降序→不排序→升序”的顺序切换。在栏标题上，会出现一个向上或向下的三角形箭头标示当前用于排序的栏，如图23-34所示。



当前按照这一栏排序

图23-34 显示在表标题栏上的三角形箭头

对单击事件的处理，是通过监听TableColumn的Selection事件实现的；而在标题栏上加箭头，则是调用了Table.setSortColumn(TableColumn)和setSortDirection(int)的结果。前者决定在哪个TableColumn上显示标记，后者则决定箭头标记的方向(SWT.UP或SWT.DOWN)。

为了记录某一栏被单击的次数，在TableColumn对象上利用setData方法存储了这一信息。初始化TableColumn时，将名为SORT_KEY的数据定义为SORT_NONE，代表不按该栏排序；当用户单击TableColumn时，程序会将这一数据改成SORT_ASC，以标志当前该栏的排序为正向排序；而第三次单击则会将该数据改为SORT_DESC，代表当前该栏会按逆序排列；如果第4次单击，该信息会回复到SORT_NONE。

下面是ViewerComparator类(代码见光盘：\book.ch23.ContactComparator2.java)。

```

public class ContactComparator2 extends ViewerComparator implements
    Listener {
    .....
    public ContactComparator2(TableViewer viewer) {
        this.viewer = viewer;
        Table table = viewer.getTable();
        int columnCount = table.getColumnCount();
        assert columnCount != 0;
    }
}

```

实现了Listener接口，用于监听


```
for (int i = 0; i < columnCount; i++) {
    table.getColumn(i).addListener(SWT.Selection, this);
    table.getColumn(i).setData(SORT_KEY, SORT_NONE);
}
```

初始化时，开始监听表格中每一个 `TableColumn` 的 `Selection` 事件。并且将它们的排序标志位设置成“不排序” (`SORT_NONE`)

```
public void handleEvent(Event event) {
    assert event.widget instanceof TableColumn;
    TableColumn column = (TableColumn) event.widget;

    int newStatus = ((Integer) column.getData(SORT_KEY) + 1) % 3;
    column.setData(SORT_KEY, newStatus);
    if (!column.equals(sortColumn)) {
        if (sortColumn != null)
            sortColumn.setData(SORT_KEY, SORT_NONE);
        sortColumn = column;
        viewer.getTable().setSortColumn(sortColumn);
    }
}
```

如果当前没有可供排序的列，则始终返回0(所有元素都相等)。由于默认采用的归并排序是一个稳定的排序算法，因此此时对象的顺序保持不变

```
int dir = 0;
switch (newStatus) {
    case SORT_ASC:
        dir = SWT.UP;
        break;
    case SORT_DESC:
        dir = SWT.DOWN;
        break;
    case SORT_NONE:
        dir = SWT.NONE;
        break;
}
```

当某一栏被点击时，如果当前不是按这一栏排序，则切换到按这一栏排序，如果当前已经是按这一栏排序了，则在升序/降序/不排序的状态间切换

调用 `Table.setSortColumn()` 和 `setSortDirection` 方法在标题处显示箭头标志(这一功能是 SWT 3.2 级中新加入的)

需要注意，这里仅仅对各个排序相关因素的状态作修改，并不直接进行排序动作。重新排序动作的触发是由最后的

```
viewer.getTable().setSortDirection(dir);
viewer.refresh();
```

最后调用查看器的 `refresh` 方法，这会触发一次重新排序动作

```
public int compare(Viewer viewer, Object e1, Object e2) {
```

```
.....
if (sortColumn == null
    || SORT_NONE == (Integer) sortColumn.getData(SORT_KEY))
    return 0;
```

```
int columnIndex = table.indexOf(sortColumn);
String a = labelProvider.getColumnText(e1, columnIndex);
String b = labelProvider.getColumnText(e2, columnIndex);
int result = StringUtils.compare(a, b);
int dir = (Integer) sortColumn.getData(SORT_KEY);
return dir == SORT_ASC ? result : -result;
```

根据所选用于排序的列，从标签提供方中取出对应的标签作为比较对象。然后调用 `StringUtils` 的 `compare` 方法比较两个字符串(这个方法支持对值为 `null` 的对象进行比较)。最后根据当前排序列的排序方式(升序/降序)对比较结果处理后返回

多列排序：即单击栏A时，表格按照栏A的值升序排列；再单击栏B时，表格会按照“栏B升序→栏A升序”的顺序排列表格内容，多次单击某一栏时，这一栏的状态同样会按照“升序→降序→不排序→升序”的顺序切换。在表格栏上，用数字和方向标明了当前的排序规则，如图23-35所示。



当前的排序顺序为: 首先按照Company栏(号码为0)

图23-35 在表的标题栏上显示排序的顺序和方向

这里对单击事件的响应与上面一致。在消息处理方法中,使用了一个栈来依次存储用户单击过的TableColumn,并根据单击次序和顺序重新排列它们,处于栈顶的是最后单击的TableColumn;排序时,按照从栈顶到栈底的顺序,取得TableColumn从标签提供方处依次取得它们对应的标签并作排序动作。

多列排序的实现代码如下所示(代码见光盘: book.ch23.ContactComparator.java)。

```
public class ContactComparator extends ViewerComparator implements Listener {
```

```
.....
```

```
private Stack<TableColumn> pushSequence;
```

该栈存储了所有将参与排序的TableColumn

```
public ContactComparator(TableViewer viewer) {
    this.viewer = viewer;
    Table table = viewer.getTable();
    int columnCount = table.getColumnCount();
    assert columnCount != 0;
    pushSequence = new Stack<TableColumn>();
    for (int i = 0; i < columnCount; i++) {
        TableColumn column = table.getColumn(i);
        column.addListener(SWT.Selection, this);
        column.setData(SORT_KEY, SORT_NONE);
        column.setData(ORIGIN_NAME, column.getText());
    }
}
```

准备工作与单列排序处相似,另外还多了将TableColumn的原始名称保存下来的一步。因为后面需要改变TableColumn的名称来标示排序的顺序,这里需要先作一个备份

```
public void handleEvent(Event event) {
    assert event.widget instanceof TableColumn;
    TableColumn column = (TableColumn) event.widget;
    int newStatus = ((Integer) column.getData(SORT_KEY) + 1) % 3;
    column.setData(SORT_KEY, newStatus);
```

在三种状态间切换

```
pushSequence.remove(column);
column.setText((String) column.getData(ORIGIN_NAME));
if (SORT_NONE != newStatus)
    pushSequence.push(column);
int length = pushSequence.size();
```

如果被单击的Column状态不为SORT_NONE,将它移到栈顶(参加排序),否则移出栈(不参加排序)

```
for (int i = length - 1; i >= 0; i--) {
    TableColumn tc = pushSequence.get(i);
    String originText = (String) tc.getData(ORIGIN_NAME);
    String dir = SORT_TEXT[ (Integer) tc.getData(SORT_KEY)];
    tc.setText(originText + " " + (length - 1 - i) + " " + dir);
}
viewer.refresh();
```

根据新的排列顺序,对各Column重新命名,然后触发查看器的刷新动作

```
public int compare(Viewer viewer, Object e1, Object e2) {
    assert viewer instanceof TableViewer;
```



```

Table table = ((TableView) viewer).getTable();
ITableLabelProvider labelProvider = ((ITableLabelProvider) ((TableView)
viewer)
    .getLabelProvider());
int length = pushSequence.size();
String[] attr1 = new String[ length];
String[] attr2 = new String[ length];
for (int i = length - 1; i >= 0; i--) {
    int columnIndex = table.indexOf(pushSequence.get(i));
    attr1[ length - 1 - i] = labelProvider
        .getColumnText(e1, columnIndex);
    attr2[ length - 1 - i] = labelProvider
        .getColumnText(e2, columnIndex);
}
for (int i = 0; i < length; i++) {
    int dir = (Integer) pushSequence.get(length - 1 - i).getData(
        SORT_KEY);
    int thisCompare = StringUtils.compare(attr1[ i], attr2[ i]);
    if (thisCompare != 0) {
        return (dir == SORT_ASC) ? thisCompare : -thisCompare;
    }
}
return 0;
}
}

```

根据表中各Column的排列顺序，从标签提供方中取得对应的标签值，存储在数组中

对数组中的值按照字典排序的原则进行排序。排序结果就是两个元素的比较结果

参照上面的示例，读者可以举一反三，实现适合自己程序需求的动态排序功能。

23.4 FAQ

本节将对初学者在实践RCP程序开发中通常会遇到的一些问题予以解答。

问：在启动RCP程序时，如何察看控制台的输出信息以及错误日志？

答：如果是在Eclipse中启动RCP程序，当程序因为错误而未能正常启动时，会显示一个错误对话框，如图23-36所示。

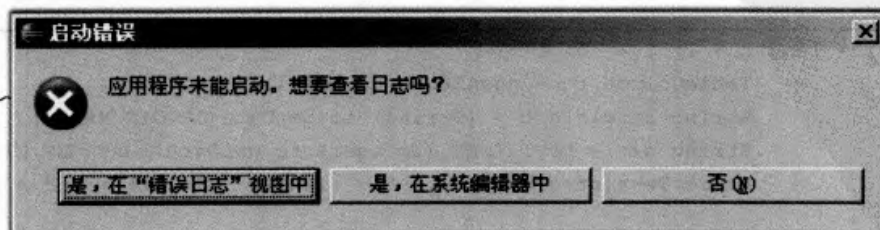


图23-36 启动RCP时的错误对话框

选择第一项“是，在‘错误日志’视图中”，Eclipse将在“错误日志”视图中列出错误；选择第二项，Eclipse会调用系统的文本编辑器（默认情况下是notepad）来打开错误日志文件，如图23-37所示。

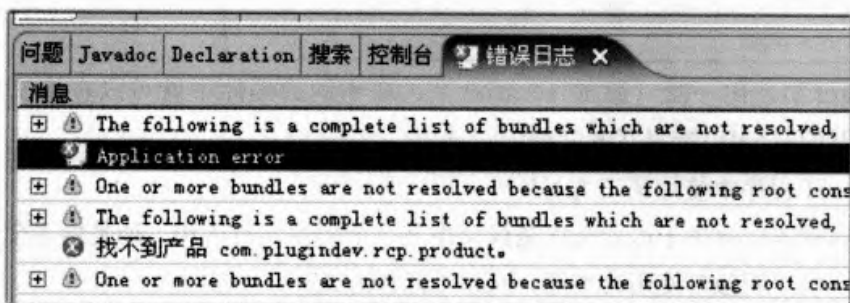


图23-37 错误日志视图

如果是独立的方式启动RCP程序，则在程序所在目录下会生成一个workspace目录。其中的“.metadata”目录下有一个“.log”日志文件。其中包含有启动RCP程序时的错误日志，如图23-38所示。

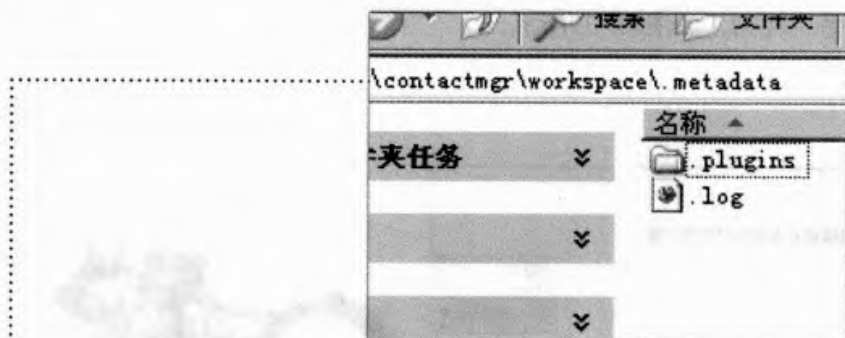


图23-38 Eclipse的错误日志



在运行RCP程序时，stdout是不可用的。这意味着用户通过System.out输出的信息都无法显示。如果需要记录调试信息或运行记录等内容，可以考虑使用各种logging框架。自JDK 1.4开始，Java中包含了一套logging框架（java.util.logging），可以很方便地实现日志记录功能。以下的代码演示了如何使用Java自带的logging功能将日志文件输出到一个文件中。

```
import java.util.logging.Logger;
..... 以普通文本的格式记载日志
Logger logger = Logger.getLogger("rcplogger");
Handler handler = new FileHandler("D:/log");
handler.setFormatter(new SimpleFormatter());
logger.addHandler(handler);
```

为Logger添加一个文件Handler，就可以把日志记录打印使用Logger记录日志，如下所示。

```
Logger logger = Logger.getLogger("rcplogger");
```

使用名称来取得一个Logger


```
logger.log(Level.INFO, "This is a log");
```

记录下来的日志文件格式如下所示。

可以为记录设置级别，按照级别可以分组记录

```
2007-8-14 17:32:37 rcpdev.contactmgr.ui.Activator start  
信息: This is a log
```

只要将Logger初始化一次（添加Handler），在不同的插件中就可以直接使用它。关于更多Logging包的使用问题，可以参见JDK文档。

问：如何在RCP程序中使用第三方类库？

答：使用类库的方式取决于这个类库是仅仅由某一个插件私有使用，还是由多个插件共同使用。

如果是某一个插件所独自使用的类库，有以下两种方式。

1. 将类库放到插件的根目录下，然后像编写普通Java项目一样，将类库加到插件项目的构建路径中。在运行时框架中，插件的类路径默认为插件的根目录，因此这样就可以保证插件载入时将类库一起读入。

2. 如果第三方类库很多，放在根目录下面会显得很乱。这种情况下可以创建一个子目录，将所有第三方类库放进去（如{根目录}/libs）。然后在“构建”选项卡的“额外的类路径条目”一项中，将这些类库添加进去。这会使Eclipse将对应的类库添加到项目类路径的“插件依赖项”列表中。其效果和手动将这个类库添加到构建路径是一样的，如图23-39所示。

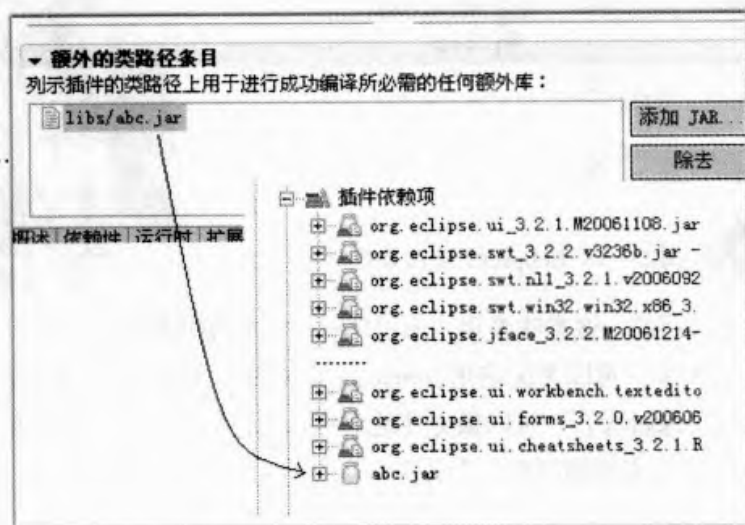


图23-39 将第三方包加入插件的依赖列表



由于现在类库不再处于插件的默认类路径下面了，因此还需要手动将它们添加进去，否则插件运行时可能会找不到这些类库。在“运行时”选项卡中，找到“类路径”的项目，单击“添加”将第三方类库添加进去，如图23-40所示。

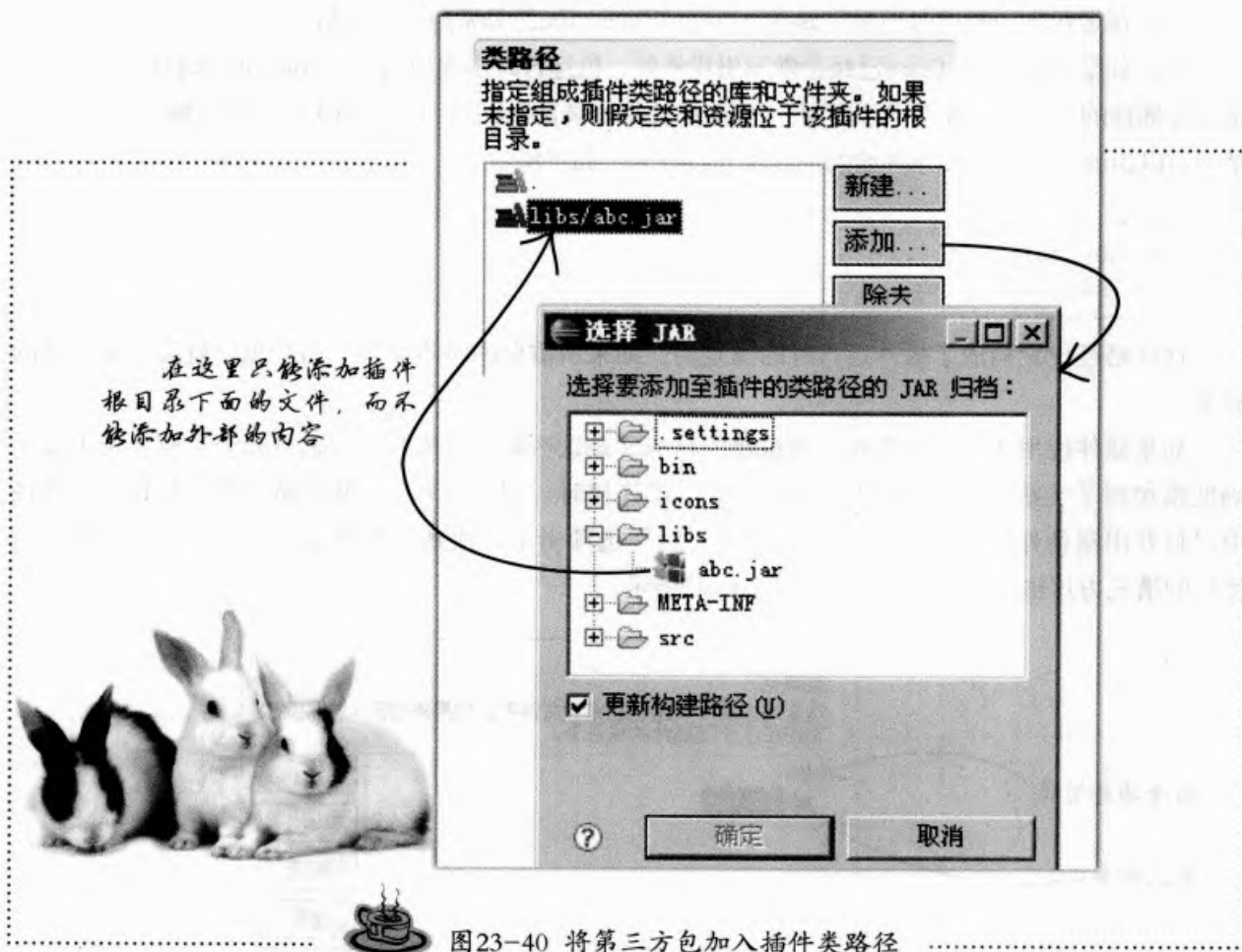


图 23-40 将第三方包加入插件类路径

如果第三方类库被多个插件同时使用，虽然也可以采用上面的方式，为每个插件重复添加一遍依赖，但是当第三方类库更新时，所有插件都需要更新，不仅烦琐而且不易于管理。因此将采用另外的方式来处理这种情况。

一种推荐的方式是创建一个单独的插件来包装这些第三方包，在插件的“运行时”选项卡中“已导出的包”一项中将第三方包中所有的内容全部导出，如图 23-41 所示，然后把这个“包裹”插件添加到所有使用这些包的插件的依赖列表中。这种做法的优点是当第三方包有更新时，只需要更新这个包裹插件，而不需要更新其他使用包的插件。

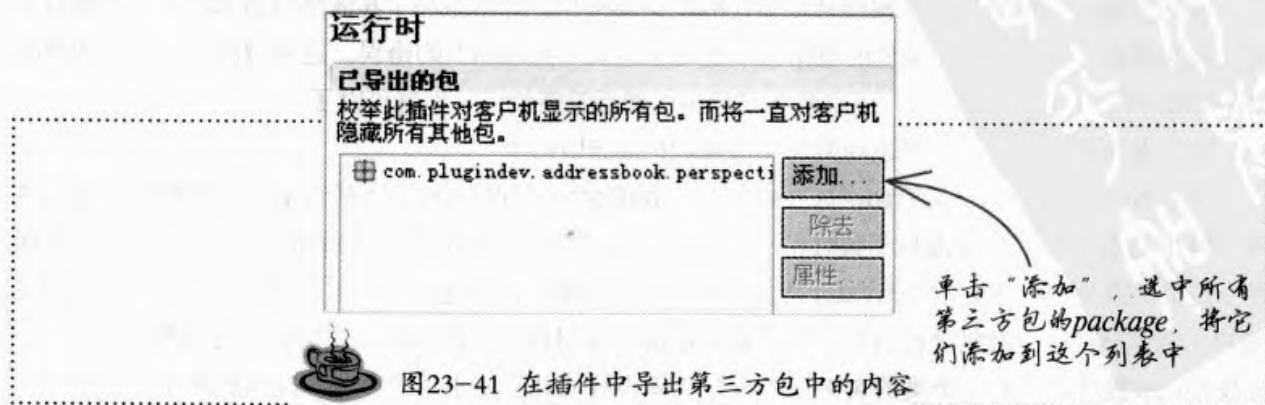


图 23-41 在插件中导出第三方包中的内容

问：在运行RCP程序时出现ClassNotFoundException的异常应该如何解决？

答：如果确定Java Class已经正常导出成插件，但运行时出现ClassNotFoundException，一般来说是在插件的类路径设置上出了问题。打开{插件根目录}/META-INF/MANIFEST.MF文件，检查是否有以Bundle-ClassPath开始的一行或几行内容，如下所示。

```
Bundle-ClassPath: .,  
libs/abc.jar,  
libs/jdbc-driver.jar
```

这些路径设置构成了插件运行时的类空间。如果所需要的类不在这些路径里，就会出现上述的异常。

如果插件使用了第三方类库，而报错的目标又是这些第三方库时，则需要检查这些库是否被正确地添加到了类路径中。如在程序中使用JDBC连接时，需要将JDBC驱动程序包添加到运行路径中。打开出错插件的plugin.xml，在“运行时”选项卡中，找到“类路径”的项目。并且保证所使用的第三方库被添加到了其中，如图23-42所示。

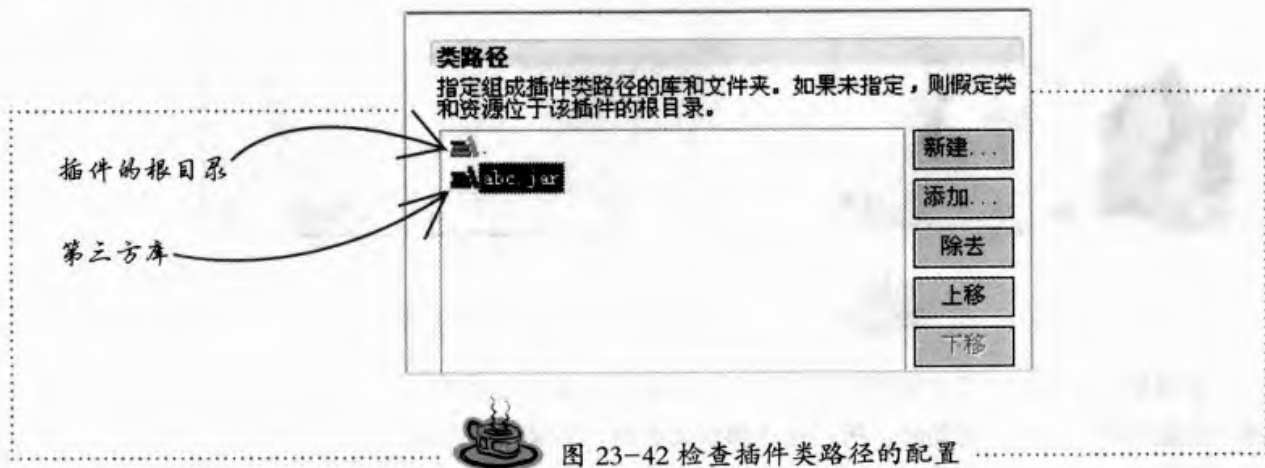


图 23-42 检查插件类路径的配置

如果插件没有使用什么第三方类库，而且异常的目标类是插件的启动器(Activator)类，那么问题很可能是由于没有将启动器所在路径，也就是插件根目录设置到类路径中。当插件被读取时，运行时框架会根据由Bundle-Activator所指定的启动器来启动插件，如果启动器所在路径不在这个类空间内，就会报出“ClassNotFound”的异常。

在没有指定其他路径时，根目录是插件的默认类路径。如果添加其他路径而又没有添加根目录时，就会将根目录“挤出”插件的类路径而出现上述找不到启动器的错误。这种情况下可以手动修改MANIFEST.MF文件，将“.”添加到类路径中，然后重新编译导出就可以了。

问：如何解决程序运行时出现的NoClassDefFoundError？

答：NoClassDefFoundError与ClassNotFoundException的区别在于后者是在运行时根据类的名称字符串动态寻找一个类而找不到时报出的异常。如使用了ClassLoader的loadClass方法或上一个问题中提到的插件根据指定的Activator类去寻找插件启动类等；在这些情况下，编译时都是不需要这些类的，它们只在运行时需要被引用；而前者则是由于JVM找不到在代码中直接引用到的类而导致的。如使用new操作符创建一个类的实例，或者在代码中导入并且使用了一个类，这些都属于直接引用。

这通常意味着某一个类在编译时存在于类路径中，而在运行时却不存在了。

在RCP程序中出现NoClassDefFound的问题，通常意味着类路径中使用到的某些第三方包没有被正确导出，首先应该到生成的RCP程序plugins目录下面找到对应的插件，检查对应的包是否在它应该在的位置，如果确认不在，则应该到plugin.xml的“构建”选项卡下面，将对应的包添加到构建目录中，然后重新导出插件，如图23-43所示。



将所用到的包添加到二进制构建路径中



图23-43 配置插件的构建内容

问：在运行RCP程序时，出现“找不到应用程序标识”的错误，应该如何解决？

答：“找不到应用程序标识”的错误常出现在修改了程序中某个插件（如添加一个扩展等）后，通过产品配置编辑器的界面试运行程序时，如图23-44所示。



图23-44 “找不到应用程序标识”的错误

这一般是由产品配置中所包含的插件不足以启动RCP程序所导致。如说插件A添加了一个扩展，所对应的扩展点包含在插件X中，而插件X却没有被加到产品配置的插件列表中；或者使用的插件X版本与它所依赖的插件不配套等。

如果使用插件配置产品，单击“添加必需的插件”就可以保证所有插件被添加。

如果使用功能部件配置产品，解决起来就相对麻烦一些。一个解决办法是将产品配置切换到按插件配置，然后单击“添加必需的插件”。在列出的插件列表中确认有哪些插件没有包含在所使用的功能部件中，再修改对应的功能部件添加它们。

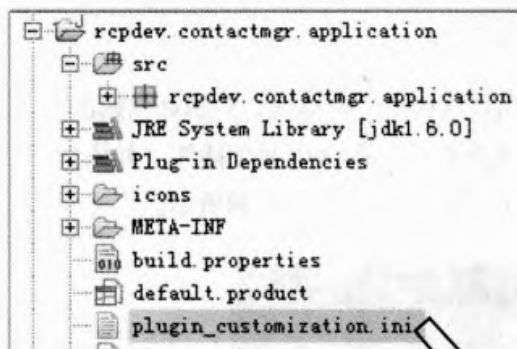
问：如何使自己的RCP程序中视图和编辑器都像Eclipse界面一样显示圆角的标题栏？

答：默认情况下，启动的RCP程序会使用方角的选项卡显示视图和编辑器的标题栏，如图23-45所示。如果不喜欢这种风格，可以选择将它调整成同Eclipse3.x一样的圆角风格。



图23-45 方角的选项卡

在包含RCP程序application扩展入口点的插件中新建一个文本文件，命名为plugin_customization.ini。并在其中加入如下内容。



```
org.eclipse.ui/SHOW_TRADITIONAL_STYLE_TABS=false
```

重新发布插件后，界面就拥有了圆角的标题栏，如图23-46所示。

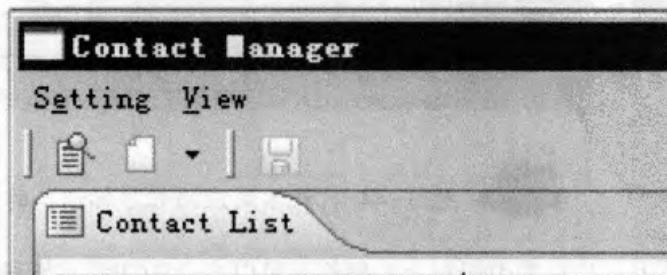



图23-46 圆角的标题栏



23.5 本章小结

本章通过概述一个从插件到RCP的开发步骤，将本书中前面部分讲到的技术重温了一遍，为读者演示了程序开发过程中从分析到设计、建模，再到最后开发、调试的步骤，并对部分Eclipse平台技术，如Job和查看器的使用做了查缺补漏，文后的FAQ摘选了部分初学者常遇到的问题并做了回答。



让我们一起沟通一下吧!

第24章 GEF实例

在GEF介绍与实现一章中已经详细了解了GEF架构组成、运行原理和创建一个GEF应用的主要步骤，本章将通过讲解创建一个实例-数据交换流程编辑器从创建工程，构建用户模型到实现控制器，绘制对应于模型的图形元素的完整过程来学习如何创建出复杂的GEF应用。本章介绍的实例中包含了使用Draw2d绘制容器控件、复杂图形控件，以及绘制立体图形效果和实现图形折叠/展开等功能。通过对本章的学习，相信读者能够举一反三，创建出令人满意的图形界面来。

本章内容包括：

- ★实例设计思路。
- ★创建GEF插件工程。
- ★创建Editor。
- ★构建模型。
- ★构建控制器。
- ★构建Figure视图。
- ★添加交互处理。
- ★创建调色板。

拉开崭新的学习帷幕

进入第24章



24.1 设计思路

数据交换流程编辑器是用来设计数据交换系统中数据流向，创建流程模板的图形化编辑工具。本章介绍的实例编辑器包括了Eclipse平台编辑器，调色板和属性框三个主要部分。为了能够完成最简单的数据库间数据流程定制功能，实例中分别实现了流程节点和数据库导入、导出节点。

流程节点是一个容器节点，它包含数据库导入、导出节点，表示一个完整的数据流程定义。一个模板中可以包含多个流程。数据库导入节点表示从数据库中将数据导入流程之中，节点下以树形结构展示数据库中的表和字段，方便用户做映射。数据库导出节点则表示把数据从流程导出到数据库。用户在添加完数据库导入、导出节点后，可以通过映射工具连接节点之间的字段，定义数据的流向。为了方便在有限的显示区域中编辑图形，这三种节点都实现了双击缩放、展开功能。在节点收缩后，连接与节点之间的映射连线会自动更新其连接位置。图24-1显示了编辑器的最终样式。

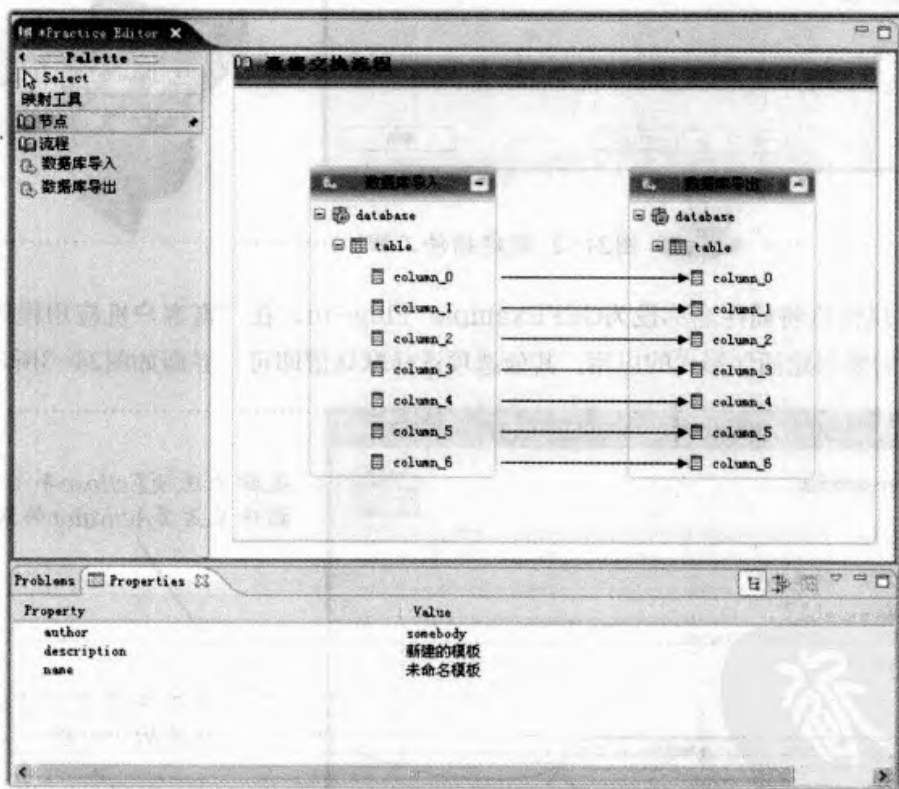


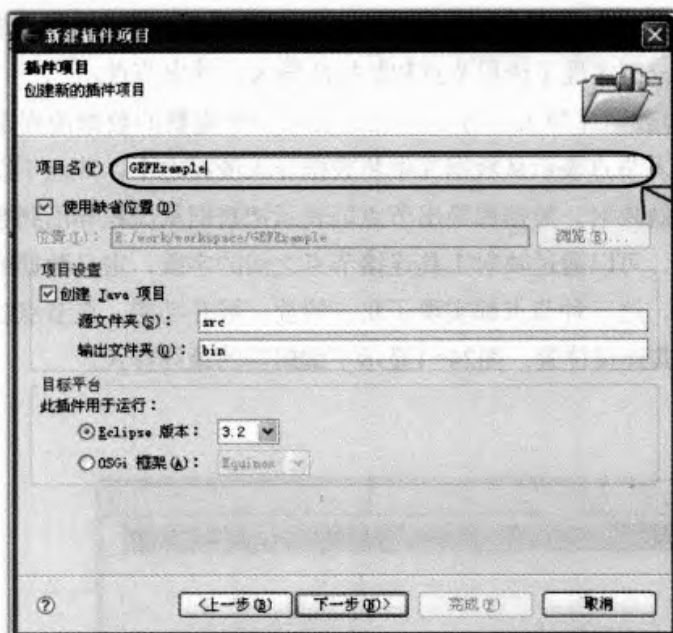
图24-1 数据交换流程编辑器

24.2 创建项目

GEF应用可以创建在Eclipse平台中的任何视图和编辑器中，因此，一个完整的GEF应用可以是一

个Eclipse插件，也可以是独立的富客户机应用程序。本章选择创建插件形式的GEF应用。

实现实例首先需要创建一个插件项目，项目创建界面如图24-2所示。实例工程创建环境是Eclipse 3.2, JDK 1.4, GEF 3.2.1。



插件工程名称设置为GEFExample



图24-2 新建插件工程

在插件内容属性页将插件名称设为GEFExample Plug-in，在“富客户机应用程序”一栏选择“否”，即表示需要创建插件形式的应用。其他选项选择默认值即可，界面如图24-3所示。



选择此项Eclipse平台将自动创建插件激活器Activator的默认实现

此选项决定了生成插件工程还是RCP工程



图24-3 设置插件工程属性

而后打开插件配置文件plugin.xml，设置插件相关属性。这一步骤在本书核心技术一章有详细介绍，这里就不赘述，但在创建GEF应用插件时需要说明的是插件依赖性属性页面中需要把org.eclipse.gef插件添加到必须的插件中，配置界面如图24-4所示。

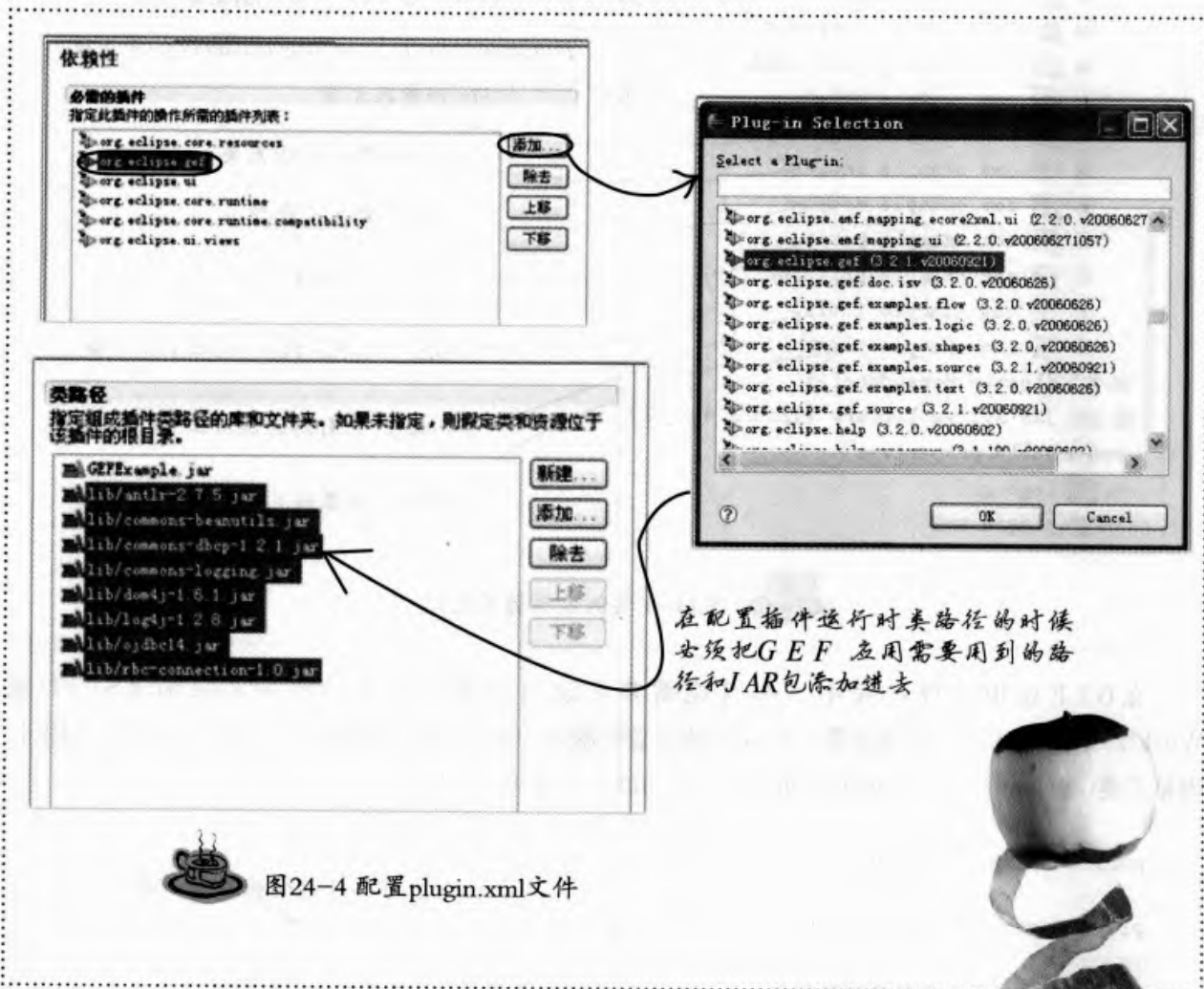


图24-4 配置plugin.xml文件

注意:



这里不需要额外添加Draw2d插件，因为Draw2d插件是GEF的依赖项，添加GEF插件后，Draw2d插件也会自动被添加进去。

初步配置完plugin.xml之后，就要创建工程中的包结构了。GEF的核心包括对应于MVC架构的模型、视图、控制器三部分，因此创建包com.sample.model放置模型相关的类，创建包com.sample.Figure放置与模型对应的图形元素Figure，创建包com.sample.EditPart放置控制器EditPart。在创建完GEF的MVC部分之后再分别按功能添加GEF应用中的其他包。图24-5显示了一个完整GEF应用工程的包结构。

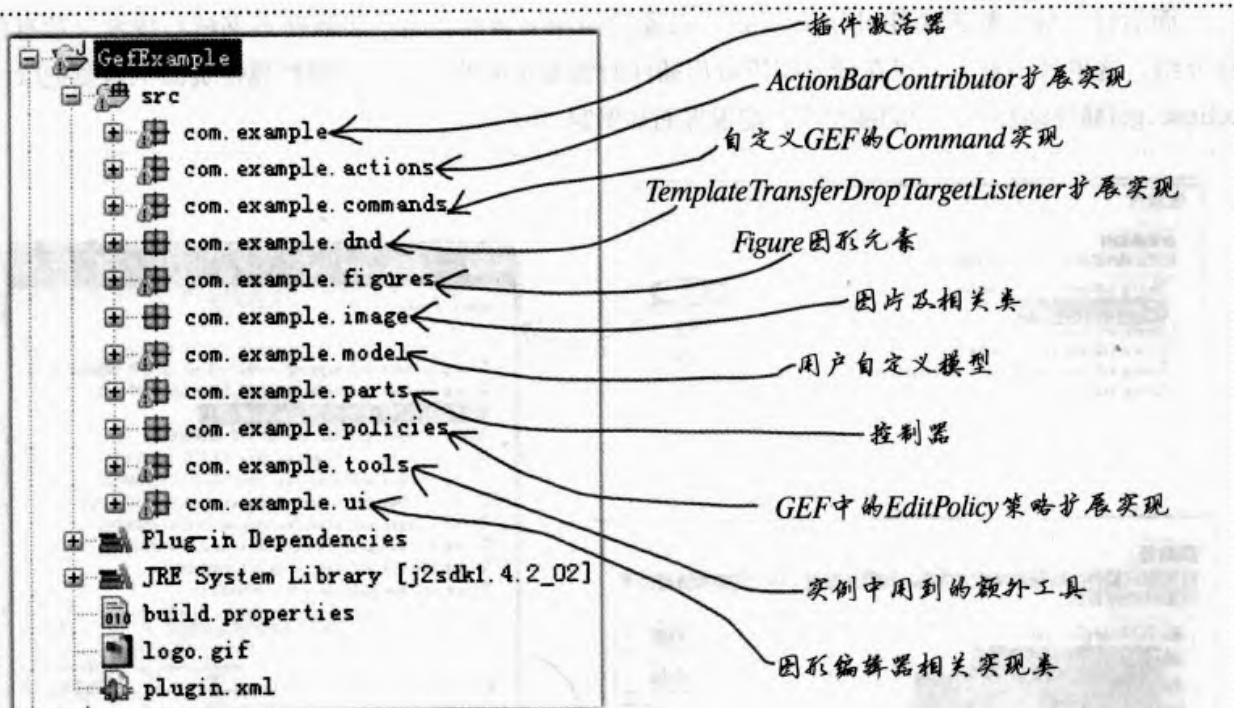


图24-5 实例编辑器的包结构

在GEF应用实现过程中，有可能需要获取当前活动状态的WorkbenchShell或WorkbenchWindow。因此需要在生成的插件激活器GefExamplePlugin的默认实现中加如下代码(代码见工程GefExample，com.example.GefExamplePlugin.java)。

```
public class GefExamplePlugin extends AbstractUIPlugin {
    private static GefExamplePlugin plugin;
    public static ImageDescriptor getImageDescriptor(String path) {
        return imageDescriptorFromPlugin(PLUGIN_ID, path);
    }
    public static Shell getActiveWorkbenchShell() {
        IWorkbenchWindow window = getActiveWorkbenchWindow();
        if (window != null) {
            return window.getShell();
        }
        return null;
    }
    public static IWorkbenchWindow getActiveWorkbenchWindow() {
        return getDefault().getWorkbench().getActiveWorkbenchWindow();
    }
}
```

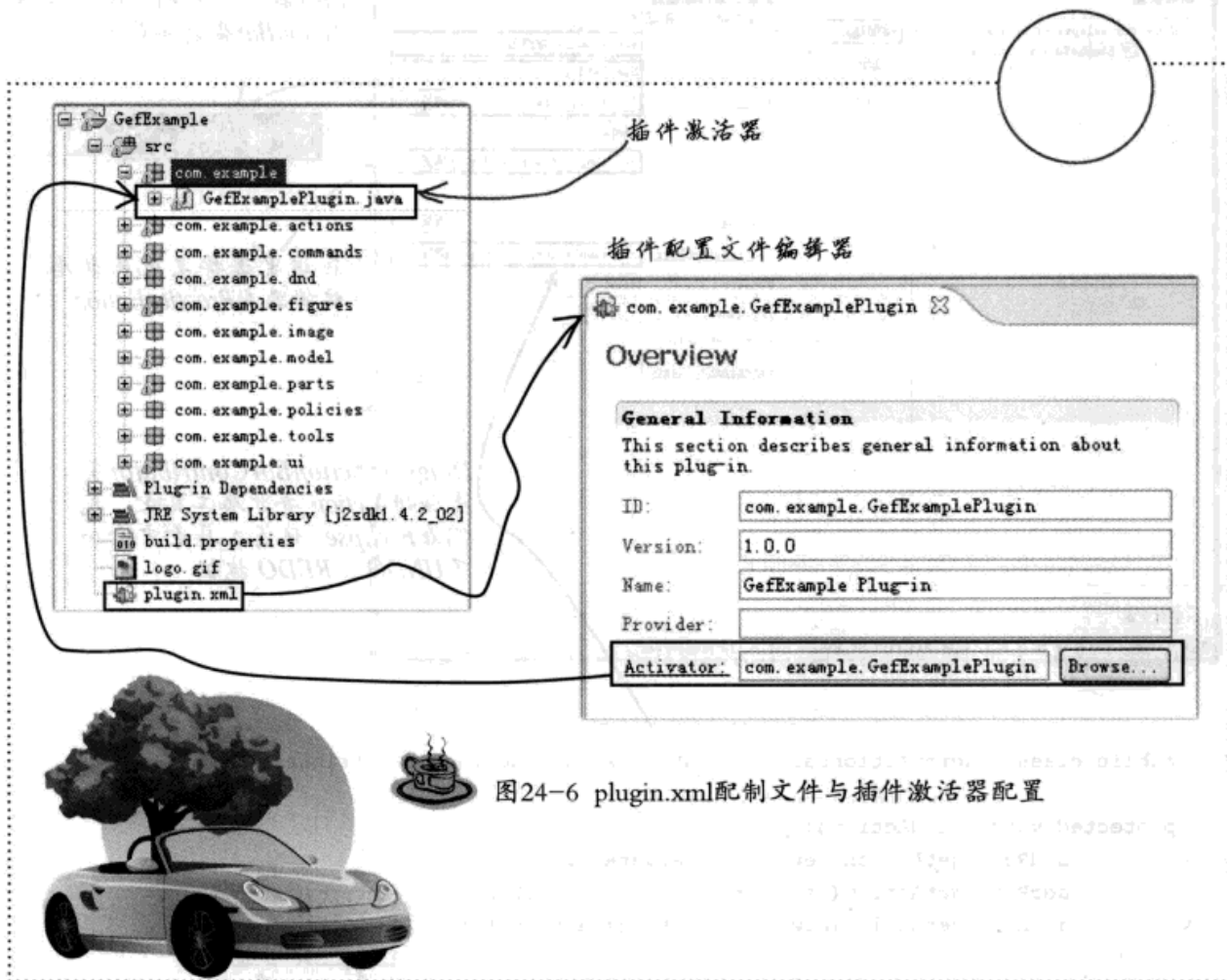
可以通过它获取当前插件实例

根据传入的路径参数获取ImageDescriptor

获取当前活动的WorkbenchShell

获取当前活动的WorkbenchWindow

默认情况下，创建向导会自动在plugin.xml配制文件的Activator中填入自动生成的插件激活器，如图24-6所示，如果用户使用了自己额外创建的激活器，就需要修改此处配置了。

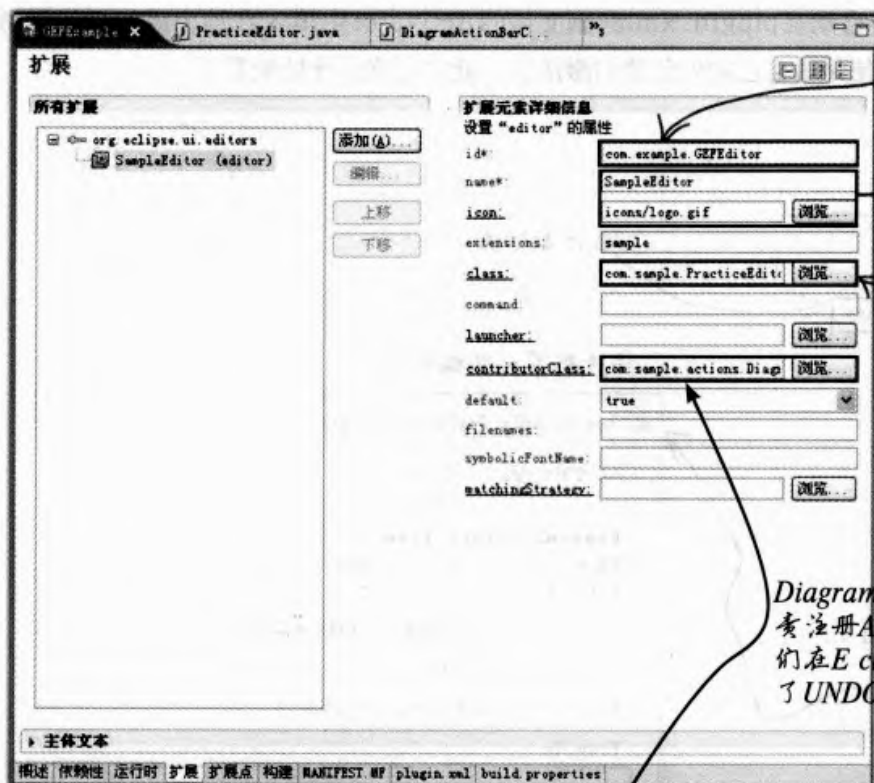


至此，一个GEF应用的插件工程框架就创建完成了。

24.3 创建Editor

编辑器Editor是GEF应用的核心组件，是绘制图形的容器，它为GEF应用提供了模型导入、保存、另存为和图形编辑是否需要保存的状态判断等方法。

Eclipse 中的Editor 是从org.eclipse.ui.part.EditorPart 扩展而来的，因为要把Editor作为操作GEF的界面，所以这里生成的DiagramEditor类是从org.eclipse.ui.part.EditorPart的子类org.eclipse.ui.parts.GraphicalEditor扩展而来。GraphicalEditor类可以帮助创建显示GEF图形的视图Viewer。在实际应用时，采用GraphicalEditorWithFlyoutPalette类，这个类是带有可以隐藏、展开并能够被拖放的调色板的GraphicalEditor的默认实现。Editor的扩展配置界面以及相关代码描述如图24-7所示。



Editor中实现了org.eclipse.ui.IEditorActionBarContributor接口, 对Eclipse平台中ActionBar做扩展的类



在这里选择实现的扩展编辑器类PracticeEditor

DiagramActionBarContributor负责注册Action并扩展工具条, 我们在Eclipse现有工具条中添加了UNDO, REDO按钮

```
public class DiagramActionBarContributor extends ActionBarContributor {
```

```
    protected void buildActions() {
        addRetargetAction(new UndoRetargetAction());
        addRetargetAction(new RedoRetargetAction());
        addRetargetAction(new DeleteRetargetAction());
    }
```

```
    protected void declareGlobalActionKeys() {
        // TODO Auto-generated method stub
    }
```

```
    public void contributeToToolBar(IToolBarManager toolBarManager) {
```

```
        toolBarManager.add(getAction(ActionFactory.UNDO.getId()));
```

```
        toolBarManager.add(getAction(ActionFactory.REDO.getId()));
```

```
    }
```

```
}
```



图24-7 配置Editor扩展

编辑器Editor是实例的主要控件, 为了实现模板的打开、编辑、保存等基本功能, 需要在其中添加初始化EditDomain和视图, 添加输入 (Input), 保存和创建PaletteRoot等代码。下面是实例中用到的编辑器PracticeEditor的主要代码实现(代码参见工程GefExample, com.example.ui.PracticeEditor.java)。

```
public class PracticeEditor extends GraphicalEditorWithFlyoutPalette{
    private FTransModel transmodel = null;
    private PaletteRoot paletteRoot;
    private ActionRegistry m_action_registry;
    private KeyHandler m_shared_key_handler;
    public static String path = null;
    public PracticeEditor() {
```

在Editor构造函数中添加设置EditDomain代码

```
setEditDomain(new DefaultEditDomain(this));
```

配置图形视图GraphicalViewer, 向视图中设置RootEditPart和EditPart工厂类-PartFactory

```
}
    protected void configureGraphicalViewer() {
```

```
        super.configureGraphicalViewer();
        ScalableFreeformRootEditPart root = new ScalableFreeformRootEditPart();
        getGraphicalViewer().setRootEditPart(root);
        getGraphicalViewer().setEditPartFactory(new PartFactory());
    }
```

```
    public void doSave(IProgressMonitor monitor) {
```

将编辑后的模板保存到文件

```
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        try {
            createOutputStream(out);
            IFile file = ((IFileEditorInput) getEditorInput()).getFile();
            file.setContents(
                new ByteArrayInputStream(out.toByteArray()),
                true, // keep saving, even if IFile is out of sync with the Workspace
                false, // dont keep history
                monitor); // progress monitor
        }
```

```
        getCommandStack().markSaveLocation();
        firePropertyChange(IEditorPart.PROP_DIRTY);
```

保存结束后需要标识CommandStack的保存位置并发送PROP_DIRTY事件, 更新界面显示的保存状态

```
    } catch (CoreException ce) {
        ce.printStackTrace();
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

返回CommandStack状态, 判断是否需要保存

```
public boolean isDirty() {
```

```
    return getCommandStack().isDirty();
}
```

设置是否允许另存为功能, 这里返回false, 不允许另存为

```
    public boolean isSaveAsAllowed() {
```

```
        return false;
    }
```

```
    protected void setInput(IEditorInput input) {
```

设置编辑器输入, 当用户双击模板文件时, 调用此方法将模板从文件中读出

```
        super.setInput(input);
    }
}
```



```

IFile file = ((IFileEditorInput) input).getFile();
ObjectInputStream in = new ObjectInputStream(file.getContents());
this.transmodel = (FTransModel) in.readObject();
in.close();

```

```
setPartName(file.getName());
```

```
} catch (IOException e) {
```

```
handleLoadException(e);
```

```
} catch (CoreException e) {
```

```
handleLoadException(e);
```

```
} catch (ClassNotFoundException e) {
```

```
handleLoadException(e);
```

```
}
```

```
}
```

```
private void handleLoadException (Exception e) {
```

```
System.err.println("*** Load failed. Using default model. ***");
```

```
e.printStackTrace();
```

```
this.transmodel = new FTransModel();
```

```
}
```

```
protected PaletteRoot getPaletteRoot() {
```

```
if (this.paletteRoot == null) {
```

```
this.paletteRoot = PaletteFactory.createPalette();
```

```
}
```

```
return this.paletteRoot;
```

```
}
```

```
public Object getAdapter(Class type) {
```

```
if (type == CommandStackInspectorPage.class)
```

```
return new CommandStackInspectorPage(getCommandStack());
```

```
if (type == ActionRegistry.class)
```

```
return getActionRegistry();
```

```
if (type == ZoomManager.class)
```

```
return getGraphicalViewer().getProperty(ZoomManager.class,
```

```
toString());
```

```
return super.getAdapter(type);
```

```
}
```

```
protected FlyoutPreferences getPalettePreferences() {
```

```
return new FlyoutPreferences() {
```

```
public int getDockLocation() {
```

```
return PositionConstants.WEST;
```

模板

打开文件后将编辑器
标题设为文件名

当模板读取出错时，新建模板

创建调色板PaletteRoot

设置对应于Eclipse平台中预定义
控件的改编器 - Adapter

设置Editor中的
调色板默认属性

```

    }
    public void setDockLocation(int location) {
    }
    public int getPaletteState() {
        return FlyoutPaletteComposite.STATE_PINNED_OPEN;
    }
    public int getPaletteWidth() {
        return 150;
    }
    public void setPaletteState(int state) {
    }
    public void setPaletteWidth(int width) {
    }
};

public void commandStackChanged(EventObject event) {
    if (isDirty()) {
        firePropertyChange(IEditorPart.PROP_DIRTY);
    }
    super.commandStackChanged(event);
}
}

```

当CommandStack发生改变时调用此方法，在此方法中添加发送IEditorPart.PROP_DIRTY事件代码以更新界面显示状态

24.4 构建模型

模型是整个编辑器的基础，也是被编辑的对象。本实例中模型的结构和控制器EditPart是一一对应的，因此，在创建模型类的时候也必须创建与之对应的EditPart。一般情况下各个模型元素会存在重叠的属性和功能，可以将这些部分抽象为抽象类。

本实例中模型存在两个抽象类FElement和FNode。FElement是所有模型的抽象类，FNode扩展了FElement类，是所有节点模型的抽象类。实例中用到的所有模型和模型之间的依赖关系如图24-8所示(代码见工程GefExample, com.example.model)。

```

public abstract class FElement implements Cloneable, Serializable, IPropertySource {
    final public static String PROP_CHANGE= "PROP_CHANGE";
    final public static String PROP_DESCRIPTION = "DESCRIPTION";
    final public static String PROP_NAME= "NAME";
    final public static String PRO_CHILD = "CHILD";
    protected String name = "untitled";
    protected String description = "";
    protected List children= new ArrayList();
}

```

```

public void addPropertyChangeListener(PropertyChangeListener l) {
    listeners.addPropertyChangeListener(l);
}

```

注册，删除属性修改监听器方法


```

}
public void removePropertyChangeListener(PropertyChangeListener l) {
    listeners.removePropertyChangeListener(l);
}

```

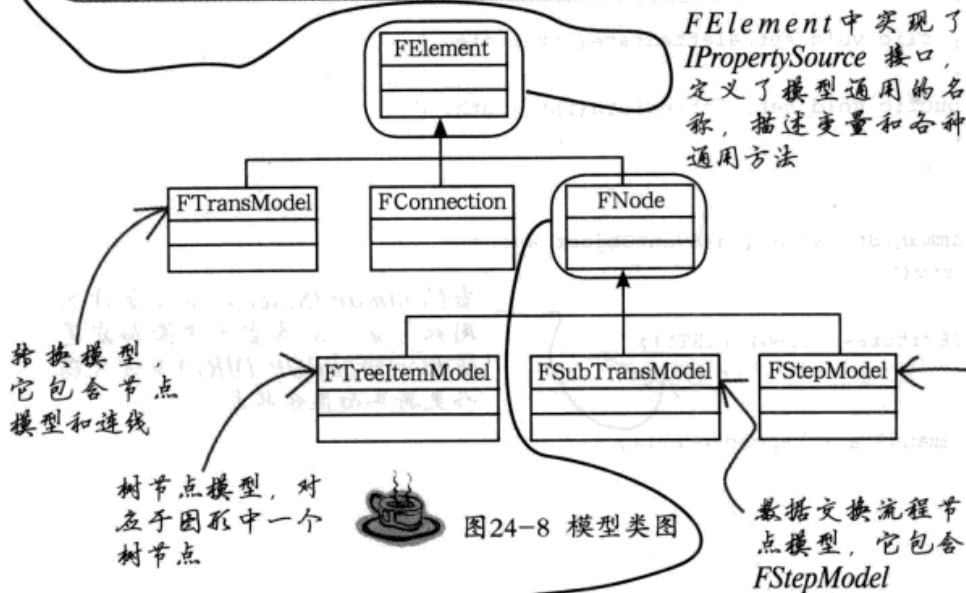
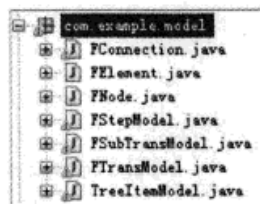
```

public void fireChildrenChange(FElement child){
    this.listeners.firePropertyChange(PRO_CHILD, null, child);
}

```

发送属性修改事件方法

*FElement*中实现了 *IPropertySource* 接口，定义了模型通用的名称，描述变量和各种通用方法



```

public abstract class FNode extends FElement {
    public static final String PRO_FIGURE = "FIGURE";
    final public static String PROP_LOCATION = "LOCATION";
    final public static String PROP_VISIBLE = "VISIBLE";
    final public static String PROP_INPUTS = "INPUTS";
    final public static String PROP_OUTPUTS = "OUTPUTS";
    final public static String PROP_SIZE = "SIZE";
}

```

```

protected Dimension size = new Dimension(100, 150);
protected Point location = new Point(0, 0);
protected boolean visible = true;
protected List outputs = new ArrayList();
protected List inputs = new ArrayList();

```

*FNode*中定义了节点模型需要用到的节点大小，位置，是否可见以及输入连线，输出连线列表

24.5 构建控制器

在GEF介绍与实现一章中已经介绍过，*EditPart*是GEF应用的控制中心，所有模型与视图之间的交互都由它来控制，因此创建*EditPart*是整个编辑器构建过程中最关键也是最复杂的部分。在数据交换流程编辑器中，希望实现数据库导入导出节点，以及节点所包含树形结构的双击展开收缩效果等复杂图形编辑功能，同时要求节点包含在流程节点内部，不允许向模板中直接添加数据库导入导出节点

点，而流程节点的大小则根据计算其中包含节点的位置和大小的结果自动设置。图24-9显示了数据交换流程节点展开和收缩的样式。

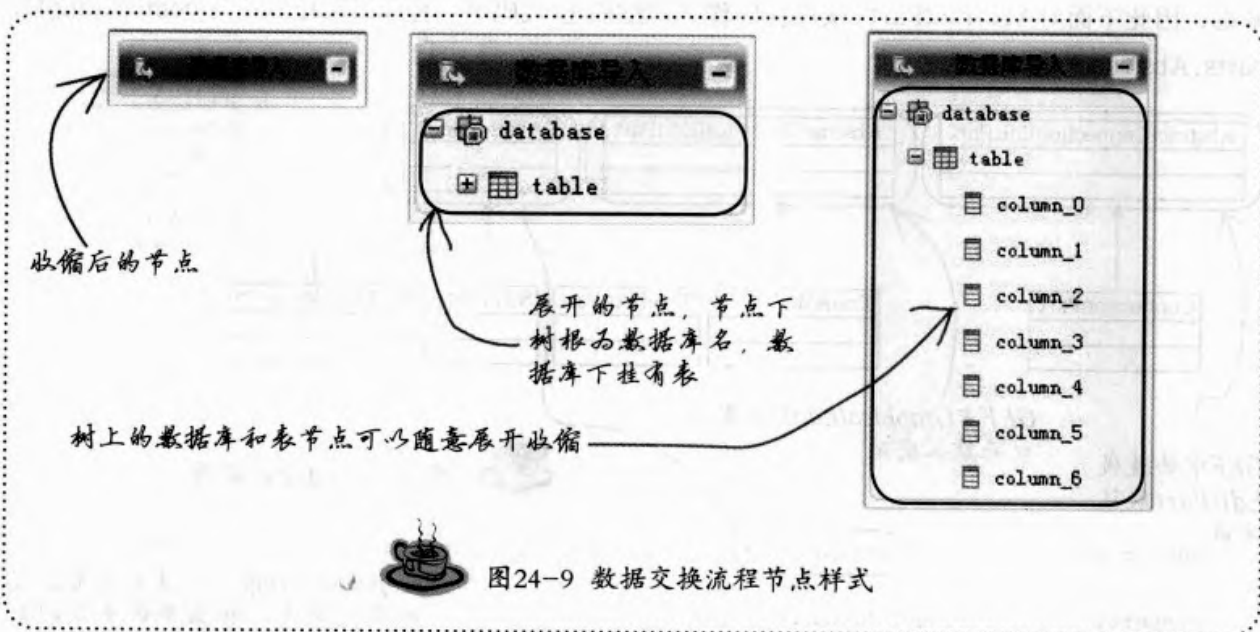


图24-9 数据交换流程节点样式

图24-10显示了在数据交换流程节点内部映射之后的数据导入导出节点的展开与收缩效果，在双击收缩节点后，连线位置也应随之变化。

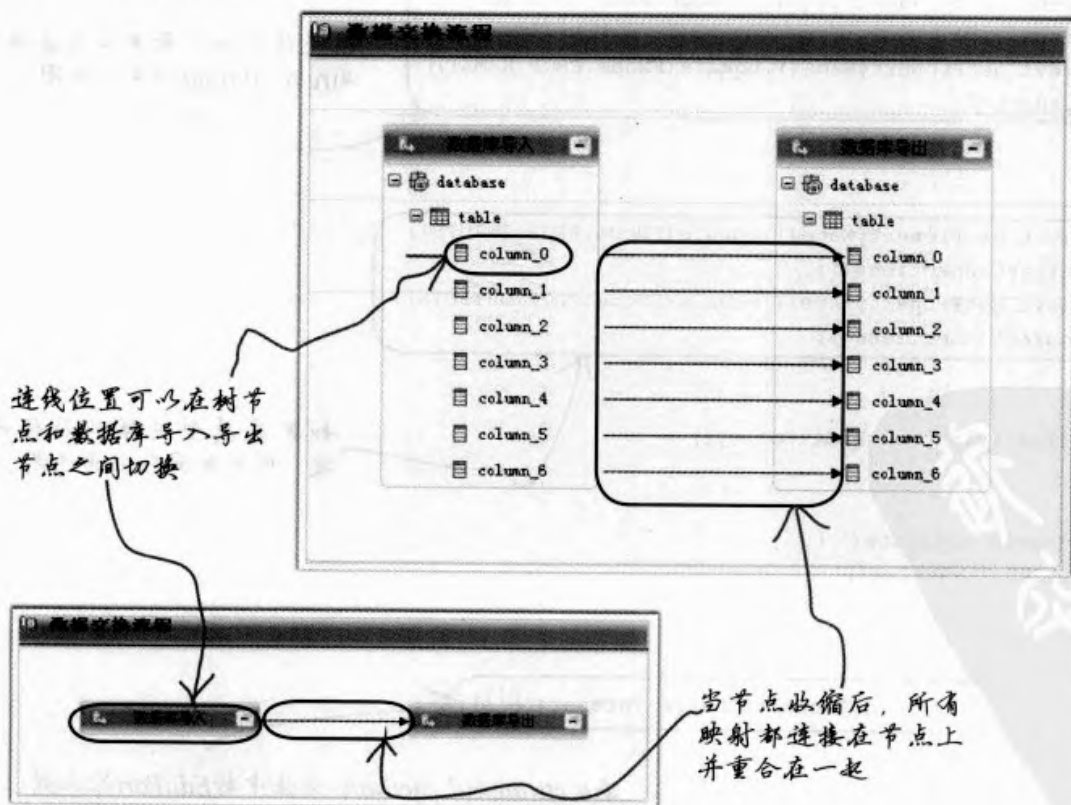


图24-10 数据交换流程节点映射

以上这些功能的实现都需要在EditPart中完成。对应已经创建好的模型，现在需要分别创建如图24-11中描述的所有EditPart。在所有需要创建的EditPart中AbstractPart是所有节点EditPart的基础，因此下面对AbstractPart 实现代码作了详细描述(代码见工程GefExample， com.example.parts.AbstractPart.java)。

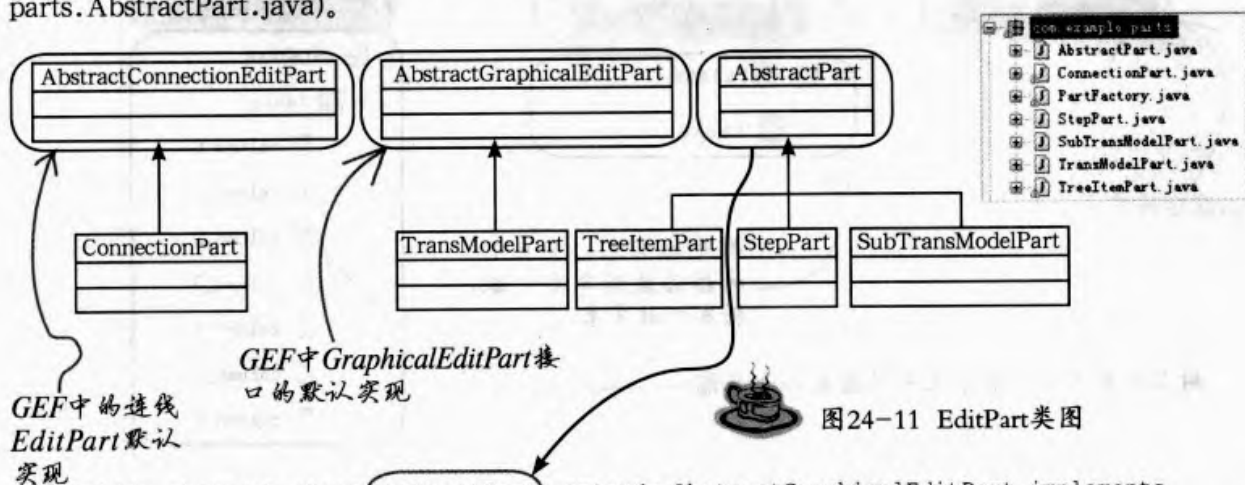


图24-11 EditPart类图

```
public abstract class AbstractPart extends AbstractGraphicalEditPart implements
```

```
PropertyChangeListener, NodeEditPart {
```

```
public void propertyChange(PropertyChangeEvent evt) {
```

```
    if (evt.getPropertyName().equals(FNode.PROP_LOCATION))
        refreshVisuals();
    else if (evt.getPropertyName().equals(FNode.PROP_NAME)){
        refreshVisuals();
    }
```

```
    else if (evt.getPropertyName().equals(FNode.PROP_INPUTS))
        refreshTargetConnections();
    else if (evt.getPropertyName().equals(FNode.PROP_OUTPUTS))
        refreshSourceConnections();
}
```

```
protected void createEditPolicy() {
```

```
public void activate() {
    if (isActive()) {
        return;
    }
}
```

```
((FNode) getModel()).addPropertyChangeListener(this);
```

```
super.activate();
```

```
public void deactivate() {
    if (!isActive()) {
```

propertyChange方法接收来自模型的修改事件，根据事件类型判断如何更新视图

模型位置和名称更改直接调用refreshVisuals()更新视图

如果是模型关联的连线修改，则更新图形上的连线

在activate和deactivate方法中将EditPart添加到模型的PropertyChangeListener或从其中移除

```

        return;
    }

    ((FNode) getModel()).removePropertyChangeListener(this);

    super.deactivate();
}

protected void refreshVisuals() {
}
}

```

在 *activate* 和 *deactivate* 方法中将 *EditPart* 添加到模型的 *PropertyChangeListener* 或从其中移除

※ 注意 ※

在实现过程中必须确保所有创建的 *EditPart* 都实现 *java.beans.PropertyChangeListener* 接口才能够监听到模型 *fire* 出来的事件。

现在已经有了模型和与之对应的控制器 *EditPart*，接下来要做的就是将模型和控制器连接在一起。连接两者的工具就是实现了 *org.eclipse.gef.EditPartFactory* 接口的 *PartFactory* 工厂类。接口 *EditPartFactory* 中定义的唯一一个方法是 *createEditPart()*，其中定义了根据模型创建对应 *EditPart* 的功能。在创建结束后还需要调用 *EditPart* 中的 *setModel()* 方法把模型设置到 *EditPart* 中。*PartFactory* 类的实现代码如下所示(代码见工程 *GefExample*， *com.example.parts.PartFactory.java*)。

```

public class PartFactory implements EditPartFactory {

    public EditPart createEditPart(EditPartContext context, Object model) {

        EditPart part = null;
        if (model instanceof FTransModel)
            part = new TransModelPart();
        else if (model instanceof FConnection)
            part = new ConnectionPart();
        else if (model instanceof FSubTransModel)
            part = new SubTransModelPart();
        else if (model instanceof TreeItemModel)
            part = new TreeItemPart();
        else
            part = new StepPart();

        part.setModel(model);
        return part;
    }
}

```

将模型设置到已经创建的 *EditPart* 中



24.6 创建Figure

Figure是在界面展现模型的图形元素，要为每一个需要显示出来的模型创建Figure。本实例中需要图形展现的模型有数据交换流程节点、数据导入导出节点、数据库树节点和连线。其中数据导入导出节点包含一颗树，需要多个Figure嵌套实现。数据交换流程节点则是带有展开折叠功能的节点容器Figure。下面首先来看一下数据交换流程节点Figure的实现。

数据交换流程节点Figure - SubTransModelFigure包含图标、标题标签、边界和一个容器Figure - SubTransContainerFigure，代码如下所示(代码见工程GefExample， com.example.Figure.SubTransContainerFigure.java)。容器Figure是包含数据库导入导出节点的Figure，因此布局采用Draw2d中提供的XYLayout布局方式。SubTransModelFigure采用ToolBarLayout布局方式。

```
public class SubTransContainerFigure extends Figure {
    public SubTransContainerFigure() {
```

```
        XYLayout layout = new XYLayout();
        setLayoutManager(layout);
```

设置XYLayout布局方式

```
        setBorder(new SubTransContainerBorder());
```

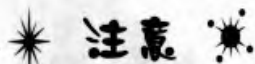
设置自定义的SubTransContainerBorder
作为Figure的边界

```
        setOpaque(false);
```

把Figure设为透明的

```
    }
```

```
}
```



注意

这里的XYLayout和ToolBarLayout布局均为org.eclipse.draw2d包中定义的布局。



SubTransModelFigure代码如下所示(代码见工程GefExample， com.example.Figure.SubTransModelFigure.java)。

```
public class SubTransModelFigure extends Figure {
    private Label name = new Label();
    private SubTransContainerFigure containerFigure = new SubTransContainerFigure();
    public SubTransModelFigure() {
```

```
        ToolbarLayout layout = new ToolbarLayout();
        layout.setVertical(true);
        layout.setStretchMinorAxis(true);
        setLayoutManager(layout);
```

设置ToolBarLayout布局方式

```
        setBorder(new LineBorder(new Color(null,211,213,220)));
```

设置指定颜色的边界

```
setOpaque(true);
```

```
name.setIcon(IconFactory.  
getImageDescriptor(ImageConstants.FRAME_LOGO).createImage());
```

```
name.setIconAlignment(Label.LEFT);
```

```
Font f = new Font(GefPracticePlugin.getActiveWorkbenchShell().getDisplay(),  
"courier", 10, 1);
```

```
name.setFont(f);
```

设置Label控件的字体

```
name.setText("数据交换流程");
```

```
name.setIconTextGap(10);
```

```
name.setLabelAlignment(PositionConstants.LEFT);
```

```
add(name);
```

以左对齐布局添加Label控件

```
add(containerFigure);
```

```
this.setOpaque(false);
```

```
}
```

```
public void paint(Graphics graphics){  
    graphics.pushState();  
    Rectangle bound = this.getBounds().getCopy();  
    bound.height -= 2;  
    bound.width -= 2;  
    graphics.setBackgroundColor(new Color(null,246,246,246));  
    graphics.setForegroundColor(new Color(null,246,246,246));  
    graphics.fillGradient(bound, false);  
    graphics.setLineStyle(SWT.LINE_SOLID);  
    graphics.setForegroundColor(new Color(null,211,213,220));  
    graphics.drawRoundRectangle(bound, 8, 8);  
    bound.height = 9;  
    graphics.setBackgroundColor(ColorConstants.lightBlue);  
    graphics.setForegroundColor(new Color(null,246,246,246));  
    graphics.fillGradient(bound, true);  
    bound.y +=9;  
    bound.height = 10;  
    graphics.setBackgroundColor(ColorConstants.lightBlue);  
    graphics.setForegroundColor(ColorConstants.lightBlue);  
    graphics.fillGradient(bound, true);  
    bound.y +=10;  
    bound.height = 9;  
    graphics.setForegroundColor(ColorConstants.lightBlue);  
    graphics.setBackgroundColor(new Color(null,246,246,246));  
    graphics.fillGradient(bound, true);  
    bound.height = 28;  
    bound.y -= 19;  
    graphics.popState();  
    super.paint(graphics);  
}
```

展开的流程节点

折叠起来的流程节点

在paint方法中，把交换流程的标题框分为了三部分，分别以蓝色渐变色绘制出立体效果



注意

在运行过程中Figure的实例是在EditPart中定义的protected IFigure createFigure()方法中创建的, 因此, 在创建完Figure类后, 还需要向这个方法中添加创建Figure实例的代码。



数据库导入导出节点是一个多层Figure嵌套的图形元素, 其中包含了由树节点TreeItemFigure多层嵌套绘制的数据库树和节点头节点, 代码如下所示(代码见工程GefExample, com.example.Figure.StepFigure.java, com.example.Figure.TreeItemFigure.java)。

```
public class StepFigure extends Shape {
    private ImageFigure downImg = null;
    private ImageFigure typeImg = null;
    public boolean ifExpand = false;
    private Label label;
```

```
private FStepModel model = null;
```

```
public class TreeItemFigure extends Shape{
    private ImageFigure expImg = null;
    private ImageFigure typeImg = null;
    private Label label = null;
```

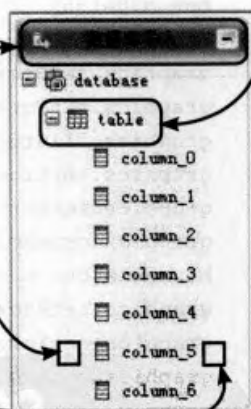
```
private TreeItemModel model = null;
```

实现中Figure 直接关联了它们所对应的模型

```
public Point getLeftAnchorLoc(){
    Point rst = Point.SINGLETON;
    rst.x = this.getBounds().x;
    rst.y = this.getBounds().y + 15;
    return rst;
}

public Point getRightAnchorLoc(){
    Point rst = Point.SINGLETON;
    rst.x = this.getBounds().x + this.getBounds().width;
    rst.y = this.getBounds().y + 15;
    return rst;
}
```

在Figure中计算映射的左右连线连接锚点位置



```
protected void outlineShape(Graphics graphics) {
    graphics.pushState();
    PointList plist = new PointList();
    Rectangle tmp = this.getBounds();
    plist.addPoint(tmp.x+6, tmp.y);
    plist.addPoint(tmp.x+6, tmp.y +10);
    plist.addPoint(tmp.x+15,tmp.y+10);
    int l = 0;
    if(!model.isIfroot()){
        TreeItemModel pa = (TreeItemModel)model.getParent();
        int ccount = pa.getChildren().size();
        if(pa.getChildren().indexOf(model) != ccount-1){
            l +=model.refreshRegion().height;
            if(ccount >1){
                PointList plist2 = new PointList();
                plist2.addPoint(tmp.x+6, tmp.y +10);
                plist2.addPoint(tmp.x+6, tmp.y +10+1);
                plist2.addPoint(tmp.x+20, tmp.y +10+1);
            }
        }
    }
    graphics.popState();
}
```

这里需要注意的是在用G raphics 绘制自己的图形时，一定要先保存Graphics 初始设置，在绘制完后再恢复Graphics的初始值，以免影响到程序其他部分的图形绘制



注意

在import 控件的时候要一定import 进正确的类，如这里就应该import org.eclipse.draw2d.Label，不要错误为其他的同名控件。



连线Figure的实现可以直接采用Draw2d中提供的PolylineConnection也可以自己创建连线实现改变连线样式。这里扩展PolylineConnection创建MappingConnection连线Figure类。由于节点和树节点都允许连线连接，因此在获取连线连接位置时BorderAnchor中的getLocation方法中还需要判断节点类型，代码如下所示(代码见工程GefExample, com.example.parts.ConnectionPart.java, com.example.Figure.MappingConnection, com.example.Figure.BorderAnchor.java, com.example.parts.StepPart.java)。

```
public class ConnectionPart extends AbstractConnectionEditPart {
    private BorderAnchor sourceAnchor;
    private BorderAnchor targetAnchor;
    protected IFigure createFigure() {
```

这里连线采用自定义的MappingConnection

```
PolylineConnection conn = new MappingConnection();
```

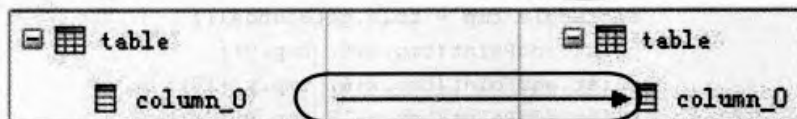


```
conn.setTargetDecoration(new PolygonDecoration());
```

设置连线端点标识

```
conn.setConnectionRouter(new BendpointConnectionRouter());
return conn;
```

设置连线路由算法

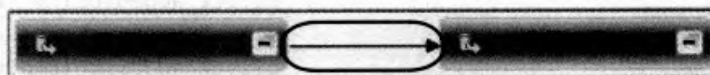


```
public class MappingConnection extends PolylineConnection {
    protected void outlineShape(Graphics g) {
        g.pushState();
```

```
g.setForegroundColor(ColorConstants.blue);
g.drawLine(this.getStart(), this.getEnd());
```

在自定义连线Figure里把连线颜色更改为蓝色，并用直线连接连线起点和终点

```
g.popState();
}
```



```
public class BorderAnchor extends ChopboxAnchor {
```

在BorderAnchor中判断连线方向和连接的节点类型后返回连接位置

```
public Point getLocation(Point reference) {
    Point p;
    p = getOwner().getBounds().getCenter();
    getOwner().translateToAbsolute(p);
    if (reference.x < p.x){
        if (getOwner() instanceof TreeItemFigure)
            p = ((TreeItemFigure)getOwner()).getLeftAnchorLoc();
        else
            p = ((StepFigure)getOwner()).getLeftAnchorLoc();
    }
    else{
        if (getOwner() instanceof TreeItemFigure)
            p = ((TreeItemFigure)getOwner()).getRightAnchorLoc();
        else
            p = ((StepFigure)getOwner()).getRightAnchorLoc();
    }
    getOwner().translateToAbsolute(p);
    return p;
}
```

```
public class StepPart extends AbstractPart {
    public ConnectionAnchor getSourceConnectionAnchor(ConnectionEditPart connection)
```

```
return new BorderAnchor(getFigure());
```

在StepPart实现的NodeEditPart接口中定义的方法中返回BorderAnchor

```
public ConnectionAnchor getSourceConnectionAnchor(Request request) {
```

```
return new BorderAnchor(getFigure());
}
```

24.7 交互处理

现在已经有了数据流程编辑器的模型、视图和控制器，运行程序就能够得到一个只有图形展现功能的编辑器。为了处理用户与编辑界面的交互，实现模型的可视化编辑功能，还需要为编辑器添加交互处理。由于EditPart是整个GEF应用的控制中心，处理交互的任务理所当然地应由EditPart承担。

在本实例程序中，需要实现的交互功能包括节点的折叠与展开，数据交换流程节点的折叠与展开，对节点树的操作以及创建、删除、移动等图形操作。下面介绍如何实现这些交互功能。

为了要让图形元素随着界面操作动起来首先需要为EditPart添加EditPolicy。前面已经介绍了EditPolicy是处理GEF请求，创建Command的类。为了能够实现默认角色Role定义的功能，必须在EditPolicy中添加相应的EditPolicy。在已经创建的EditPart中还需要分别添加以下代码(代码见工程GefExample, com.example.parts)。

```
public class SubTransModelPart extends AbstractPart {  
    protected void createEditPolicy() {
```

```
        installEditPolicy(EditPolicy.LAYOUT_ROLE, new SubTransLayoutEditPolicy());  
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new NodeEditPolicy());
```

```
    }
```

```
public class StepPart extends AbstractPart {  
    protected void createEditPolicy() {
```

```
        installEditPolicy(EditPolicy.COMPONENT_ROLE,  
            new StepEditPolicy());
```

```
    }
```

```
public class TreeItemPart extends AbstractPart {  
    protected void createEditPolicy() {
```

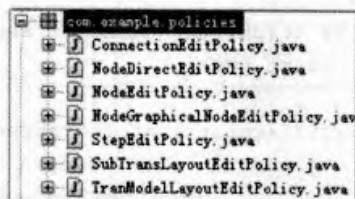
```
        TreeItemModel model = (TreeItemModel)this.getModel();  
        if(model.getType() == TreeItemModel.TYPE_COLUMN){  
            installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new  
                NodeGraphicalNodeEditPolicy());  
        }
```

```
    }
```

```
public class TransModelPart extends AbstractGraphicalEditPart  
    implements PropertyChangeListener {  
    protected void createEditPolicy() {
```

```
        installEditPolicy(EditPolicy.LAYOUT_ROLE, new TranModelLayoutEditPolicy());
```

```
    }
```



树节点EditPart在添加处理连线相关操作的GraphicalNodeEditPolicy时还需要判断是否是字段节点，只有字段节点才能连线


```
public class ConnectionPart extends AbstractConnectionEditPart {
    protected void createEditPolicy() {
```

```
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new ConnectionEditPolicy());
        installEditPolicy(EditPolicy.CONNECTION_ENDPOINTS_ROLE, new
            ConnectionEndpointEditPolicy());
    }
```

当创建数据库导入导出节点的时候，只能允许用户把数据库导入导出节点添加到数据交换流程节点内部，而数据交换流程节点也只能被创建在模型内部。为了实现这项功能，需要在TranModelLayoutEditPolicy和SubTransLayoutEditPolicy类中做些修改，代码如下所示(代码见工程GefExample, com.example.policies.TranModelLayoutEditPolicy.java, com.example.policies.SubTransLayoutEditPolicy.java)。

```
public class TranModelLayoutEditPolicy extends XYLayoutEditPolicy {
```

```
    protected Command createAddCommand(EditPart child, Object constraint) {
        return null;
    }
```

*XYLayoutEditPolicy*是用于处理使用XYLayout布局的Figure的界面操作

```
    protected Command createChangeConstraintCommand(EditPart child, Object constraint) {
```

```
        if (!(child instanceof SubTransModelPart))
            return null;
```

这里的constraint为org.eclipse.draw2d.geometry.Rectangle

```
        if (!(constraint instanceof Rectangle))
            return null;
```

创建移动流程节点的Command，这里只负责处理流程节点，不关心其他节点

```
        MoveNodeCommand cmd = new MoveNodeCommand();
        cmd.setNode((FNode) child.getModel());
        cmd.setLocation(((Rectangle) constraint).getLocation());
        return cmd;
    }
```

```
    protected Command getCreateCommand(CreateRequest request) {
```

```
        if (request.getNewObject() instanceof FSubTransModel) {
```

创建新建流程节点的Command

```
            CreateNodeCommand cmd = new CreateNodeCommand();
            cmd.setDiagram((FTransModel) getHost().getModel());
            cmd.setNode((FNode) request.getNewObject());
            Rectangle constraint = (Rectangle) getConstraintFor(request);
            cmd.setLocation(constraint.getLocation());
            return cmd;
        }
```

```
        return null;
    }
```

```
    protected Command getDeleteDependantCommand(Request request) {
        return null;
    }
```

```
}
```

```
public class SubTransLayoutEditPolicy extends XYLayoutEditPolicy {
    protected Command createAddCommand(EditPart child, Object constraint) {
        return null;
    }
    protected Command getCreateCommand(CreateRequest request) {
        Object obj = request.getNewObject();
        if (!(obj instanceof FStepModel))
            return null;

        CreateStepCommand cmd = new CreateStepCommand();
        cmd.setParent((FSubTransModel) getHost().getModel());
        cmd.setChild((FStepModel) request.getNewObject());
        Rectangle constraint = (Rectangle) getConstraintFor(request);
        cmd.setLocation(constraint.getLocation());
        return cmd;
    }
}
```

流程节点的SubTransLayoutEditPolicy中只允许创建数据库导入导出节点

为各个EditPart添加了EditPolicy之后，程序已经能够在界面上实现创建、删除、移动等针对各个节点图形元素的操作了。下面需要做的就是实现数据交换流程节点和数据库导入导出节点的展开收缩功能了。节点展开收缩功能主要为了方便图形的展现和编辑，不需要Undo/Redo功能，所以不采用创建Command的方式实现。以下是节点EditPart中相关的核心代码实现(代码见工程GefExample, com.example.parts)。

```
public class StepPart extends AbstractPart {
    public void performRequest(Request req) {
        if (req.getType().equals(RequestConstants.REQ_OPEN)){
            expandAction();
        }
    }

    private void expandAction(){
        FStepModel step = (FStepModel)this.getModel();
        if (step.isExpand()){
            step.setExpand(false);
            step.getRoot().hideAllChildVsb();
            step.getRoot().hide();
            step.collapse();
            this.refreshAll();
        } else{
            step.setExpand(true);
            step.getRoot().showAllChildVsb();
            step.getRoot().show();
            step.expand();
        }
    }
}
```

REQ_OPEN请求是GEF的RequestConstants接口中定义的双击目标图形元素产生的，它可以对应弹出对话框，打开属性编辑器等相关自定义动作，这里用它来实现双击展开，收缩节点功能

方法expandAction根据模型当前记录的状态设置自身和所属于模型的属性，最后更新视图


```

        this.refreshAll();
    }
}

public class SubTransModelPart extends AbstractPart {
    public void performRequest(Request req) {
        if (req.getType() == RequestConstants.REQ_OPEN){
            openAction();
        }
    }
}

```

```

private void openAction(){

```

流程节点的展开收缩实现与数据库导入导出节点类似

流程节点收缩后隐藏掉节点内包含的所有子节点，在展开节点时展现

```

getSubject().setCollapsed(!getSubject().isCollapsed());
for(Iterator iter = ((FSubTransModel)this.getModel()).
getChildren().iterator();iter.hasNext();){
    FStepModel step = (FStepModel)iter.next();
    if(getSubject().isCollapsed()){
        step.getRoot().hide();
        step.getRoot().hideAllChildVsb();
        ifCollapse = true;
    }else{
        step.getRoot().show();
        step.getRoot().showAllChildVsb();
        ifCollapse = false;
    }
}

```

```

for(Iterator iter = this.getChildren().iterator();iter.hasNext();){
    StepPart step = (StepPart)iter.next();
    step.refreshAll();
}

```

更新流程节点中包含的所有数据库导入导出节点视图



在实现节点收缩功能的过程中，需要连线位置也随着节点收缩而改变，因此，还需要在树节点、数据库导入导出节点的EditPart实现和TreeItemModel模型中添加以下代码(代码见工程GefExample, com.example.parts.TreeItemPart.java, com.example.parts.StepPart.java, com.example.model.TreeItemModel.java)。

```
public class TreeItemPart
extends AbstractPart {
protected List
```

```
getModelSourceConnections() {
```

```
return
getSubject().getShowOutgoingConnections();
}
protected List
```

```
getModelTargetConnections() {
```

```
return getSubject().getShowIncomingConnections();
}
```

```
public class StepPart extends AbstractPart {
```

```
protected List getModelSourceConnections() {
```

```
if(((FStepModel)this.getModel()).isExpand()
|| ((SubTransModelPart)this.getParent()).getState())
return Collections.EMPTY_LIST;
return getStepRoot().getAllOutgoings();
}
```

```
protected List getModelTargetConnections() {
```

```
if(((FStepModel)this.getModel()).isExpand()
|| ((SubTransModelPart)this.getParent()).getState())
return Collections.EMPTY_LIST;
return getStepRoot().getAllIncomings();
}
```

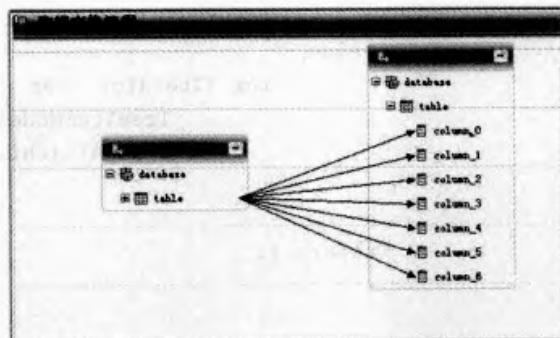
```
public class TreeItemModel extends FNode {
```

```
public List getAllIncomings(){
```

```
List l = new ArrayList();
if(this.type == TreeItemModel.TYPE_COLUMN){
return(this.getIncomingConnections());
}else{
for (Iterator iter = this.getChildren().iterator(); iter.hasNext();){
TreeItemModel child = (TreeItemModel) iter.next();
l.addAll(child.getAllIncomings());
}
}
return l;
}
```

```
public List getAllOutgoings(){
```

```
List l = new ArrayList();
if(this.type == TreeItemModel.TYPE_COLUMN){
```



这两个方法是AbstractGraphicalEditPart中定义的返回所有起点，终点连接到该节点的连线，GEF会根据这两个方法的返回值绘制连线

返回所有终点连接在树字段类型节点的连线模型

返回所有起点连接在树字段类型节点的连线


```

        return(this.getOutgoingConnections());
    } else {
        for (Iterator iter = this.getChildren().iterator(); iter.hasNext();) {
            TreeItemModel child = (TreeItemModel) iter.next();
            l.addAll(child.getAllOutgoings());
        }
    }
    return l;
}

```



24.8 创建调色板

调色板是管理各种工具的容器，它负责管理各种图形编辑工具。如果编辑器中需要用到的工具比较多，通常会把这些工具放置在调色板中，当用户编辑图形的时候可以选择调色板中的工具。本实例中采用了可以被收缩展开并能被拖动的调色板，要在GEF应用中实现这种功能的调色板，只需使编辑器PracticalEditor扩展GEF中提供的默认编辑器实现GraphicalEditorWithFlyoutPalette，而后在getPaletteRoot()方法中添加创建PaletteRoot对象的代码，如下所示(代码见工程GefExample, com.example.ui.PracticeEditor.java, com.example.tools.PaletteFactory.java)。

```

public class PracticeEditor extends GraphicalEditorWithFlyoutPalette {
    protected PaletteRoot getPaletteRoot() {
        if (this.paletteRoot == null) {
            this.paletteRoot = PaletteFactory.createPalette();
        }
        return this.paletteRoot;
    }
}

```

由调色板工厂创建调色板并返回PaletteRoot

```

public class PaletteFactory {
    public static PaletteRoot createPalette() {
        PaletteRoot paletteRoot = new PaletteRoot();
        paletteRoot.addAll(createCategories(paletteRoot));
        return paletteRoot;
    }

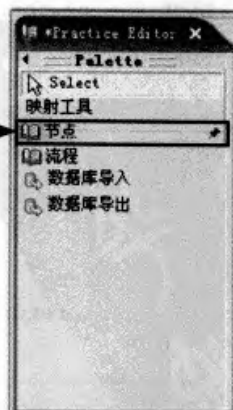
    private static List createCategories(PaletteRoot root) {
        List categories = new ArrayList();

        categories.add(createControlGroup(root));
        categories.add(createComponentsDrawer());

        return categories;
    }

    private static PaletteContainer createControlGroup(PaletteRoot root) {

```



调色板共分两个分组，一个是通用工具分组，另一个是自定义可折叠的PaletteDrawer，需要在其中添加对应于节点模型的ToolEntry

```

PaletteGroup controlGroup = new PaletteGroup("工具");
List entries = new ArrayList();
ToolEntry tool = new SelectionToolEntry();
entries.add(tool);
root.setDefaultEntry(tool);
tool = new ConnectionCreationToolEntry("映射工具", "创建映射", null, null, null);
entries.add(tool);
controlGroup.addAll(entries);
return controlGroup;
}

private static PaletteContainer createComponentsDrawer() {
    PaletteDrawer drawer = new PaletteDrawer("节点");
    List entries = new ArrayList();
    ToolEntry tool = new CombinedTemplateCreationEntry("流程", "流程",
FSubTransModel.class, new SimpleFactory(
        FSubTransModel.class), null, null);
    ImageDescriptor imageDescriptor = IconFactory.
getImageDescriptor(ImageConstants.FRAME_LOGO);
    tool.setSmallIcon(imageDescriptor);
    tool.setLargeIcon(imageDescriptor);
    entries.add(tool);
    tool = new CombinedTemplateCreationEntry("数据库导入", "把数据导入数据库",
FStepModel.class, new StepFactory(
        FStepModel.class,"数据库导入"), null, null);
    imageDescriptor = IconFactory.
getImageDescriptor(ImageConstants.STEP_DB_INPUT_ICON);
    tool.setSmallIcon(imageDescriptor);
    tool.setLargeIcon(imageDescriptor);
    entries.add(tool);
    tool = new CombinedTemplateCreationEntry("数据库导出", "把数据导出数据库",
FStepModel.class, new StepFactory(
        FStepModel.class,"数据库导出"), null, null);
    imageDescriptor = IconFactory.
getImageDescriptor(ImageConstants.STEP_DB_INPUT_ICON);
    tool.setSmallIcon(imageDescriptor);
    tool.setLargeIcon(imageDescriptor);
    entries.add(tool);
    drawer.addAll(entries);
    imageDescriptor = IconFactory.
getImageDescriptor(ImageConstants.FRAME_LOGO);
    drawer.setSmallIcon(imageDescriptor);
    drawer.setLargeIcon(imageDescriptor);
    return drawer;
}

public class StepFactory extends SimpleFactory {

```

StepFactory 扩展自 GEF 中默认提供的创建实例的工厂类 SimpleFactory，它实现了 CreationFactory 接口，当用户选中该工具时，依赖 StepFactory 创建模型实例


```

public String step_name = null;

public StepFactory(Class aClass, String name) {
    super(aClass);
    step_name = name;
}

public Object getNewObject() {
    FStepModel step = (FStepModel)super.getNewObject();
    if(step_name != null)
        step.setName(step_name);
    return step;
}

```



实例中，数据库导入节点和数据库导出节点用一个模型实现，两节点的唯一差别就是名称，因此在创建模型对象实例的`CreationFactory`类`StepFactory`中添加静态变量`step_name`，当节点对象创建结束后，把节点名称设置为`step_name`

24.9 本章小结

至此，完成了整个GEF实例程序——数据流程编辑器的创建。GEF是一个功能强大的创建图形化编辑模型的工具框架，通过它不但可以轻松创建如 workflow 定义工具、页面流定义工具等各种流程定义工具，甚至可以创建出报表定义工具、表单定义工具等其他图形编辑工具。在实际应用中，可以根据需求灵活运用架构中提供的功能，相信能够满足大多数图形编辑方面的需求。



轻松学会了本书内容！耶！

[G e n e r a l I n f o r m a t i o n]

书名= E c l i p s e 插件开发学习笔记

作者= 张鹏等编著

页码= 6 5 6

I S B N = 6 5 6

S S 号= 1 2 0 0 8 1 3 3

d x N u m b e r = 0 0 0 0 0 6 5 6 4 4 8 6

出版时间= 2 0 0 8

出版社= 该引擎未能查询到

定价:

试读地址= <http://book.szdnnet.org.cn/bookDetail.jsp?dxNumber=000006564486&d=E9D3A718ECFF7074E15A4D52B3D221E7&fenlei=18170403010205&sw=Eclipse>

全文地址= <http://img7.5read.com/image/ss2jpg.dl1?did=b38&pid=267DB0D1DDE0C952EE5C2BFAF9B1DCB902D5AEBD159C883B7527FF703AA4F7979D754EAF78042A264E16A02E8C4AABAC66B46A4655D7499AD8BA1E1A358D8B16895988598AAEE6B04233954FDEBECDD255720B99C36130EEABFBBC2750A4FCF9514BEBD48F88E7A93023873DC3F3FB32D7DEC33&jid=/>

Java学习群

72030155

进群可以免费获取视频教程以及每日免费听老师讲课